2006-09

A new framework for software visualization a multi-layer approach

Spyrou, Dimitrios.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A NEW FRAMEWORK FOR SOFTWARE VISUALIZATION: A MULTI-LAYER APPROACH

by

Dimitrios Spyrou

September 2006

Thesis Advisor:	Thomas Wu Otani
Second Reader:	Man-Tak Shing

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2006	Master's Thesis

4. TITLE AND SUBTITLE A New Framework for Software Visualization: A Multi-Layer Approach	5. FUNDING NUMBERS
6. AUTHOR(S) Dimitrios Spyrou	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	

13. ABSTRACT (maximum 200 words)

Software Visualization can play a significant role in our fight against Software's inherent complexity. Despite all efforts made so far, Software Visualization tools have not succeeded to be a part of Software Engineer's everyday practice. We believe that a properly defined taxonomy that will provide a framework for discussion, analysis and research guidance by offering a systematic and systemic overview of the area, covering all the concerns and challenges, is a starting point for a new approach for the field.

After analyzing existing taxonomies and exploring existing tools, we approach Software Visualization as an interface between humans and software and we propose a multi-layered framework that incorporates all the concerns and the challenges of our field, in a neat, systematic and expandable way that can also serve as a roadmap for a research area and that can promote communication of existing and new ideas.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Visualization, taxonomy, Framework, Multi-Layer Approach, Tools' Integration, Software Visualization Definition	145
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution unlimited

A NEW FRAMEWORK FOR SOFTWARE VISUALIZATION: A MULTI-LAYER APPROACH

Dimitrios Spyrou
Lieutenant Commander, Hellenic Navy
B.S., Hellenic Naval Academy, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Dimitrios Spyrou

Approved by: Thomas Wu Otani
Thesis Advisor

Man-Tak Shing
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software Visualization can play a significant role in our fight against Software's inherent complexity. Despite all efforts made so far, Software Visualization tools have not succeeded to be a part of Software Engineer's everyday practice. We believe that a properly defined taxonomy that will provide a framework for discussion, analysis and research guidance by offering a systematic and systemic overview of the area, covering all the concerns and challenges, is a starting point for a new approach for the field.

After analyzing existing taxonomies and exploring existing tools, we approach Software Visualization as an interface between humans and software and we propose a multi-layered framework that incorporates all the concerns and the challenges of our field, in a neat, systematic and expandable way that can also serve as a roadmap for a research area and that can promote communication of existing and new ideas.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENT

This thesis is dedicated to all those people who believed in me and supported me the last four years; from the moment I decided to apply for a Master's Degree at the Naval Postgraduate until now, but also, to all those who tried hard, but did not succeed, to discontinue my wonderful trip towards knowledge.

I would like to acknowledge my advisor Dr. Thomas Otani, who was always standing by my side, provided me with the opportunity to pursue my own research path letting me work in my own pace, being very patient in periods of slow progress, understanding that I was novice in the area of Software Visualization, always providing helpful critiques revealing the insights of our research object and most importantly, never stopped believing in me. His help was invaluable.

Many thanks to Dr. Man-Tak Shing, who offered his time and knowledge for the completion of this work and to my brother Dr. Thomas Spyrou who was always there for me; every single discussion with him was a broadening of my horizons.

Many thanks to Commander Dionysios Antonopoulos HN; without his help, things would be very different.

Last and more important, I must single out the encouragement I received from my fellow-traveler in this life; my wife, Mary. Despite being neglected many times during my countless working hours and the distance that came in between us during the last two months, she was constantly supporting me with her love, trust, and patient, always encouraging me to keep on while simultaneously raising my two wonderful kids.

God bless them all.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

The field of software engineering concerns itself with the technology and processes of software development, and thus it has approached the problems of software complexity and incomprehensibility in a number of ways.

Software development techniques such as top-down design and stepwise refinement [1], structured programming [2], modularity and information hiding [3], object oriented programming [4] and design patterns [5] are only a small number of the existing ones that assist Software engineers in their difficult task.

Another significant and promising approach is the use of CASE (computer-aided software engineering) tools such as rapid prototyping tools or visual tools. More specifically, Visual Programming and Program Visualization were always a “wish” as an aid for experts to anticipate software’s complexity and have been part of the software engineering culture from its inception. It’s obvious that, as software projects will continue to grow in complexity and size, tools that aid teams to understand, design, program, debug, maintain and evolve will always be welcome. Here is the call for Software Visualization that has the potential to contribute significantly to the development of software products by providing solutions to many of the problems software engineers, software project leaders, and programmers are facing. This is the reason why Software Visualization has rapidly emerged as an important field in Computer Science and Software Engineering and research is being conducted into this area globally.

B. OBJECTIVES

Various Software Visualization tools have been touted as panaceas for many Software Engineering problems while exaggerated claims are

commonplace in both academia and commercial Software Visualization tools. Despite the significant efforts made by various researchers worldwide, visual tools have not been integrated into the mainstream and professional developers still work in much the same way as they did in the early days of the computing era.

We believe that there is great potential in this area and this thesis' focus is to propose a new framework that will assist in the developing of more effective tools that will amplify the cognition of the various software artifacts, to all the participants in the development process.

C. RESEARCH QUESTIONS

Our primary target is an exploration of the current state of the area so that we can discover the challenges and propose solutions for them. The study addresses the following research questions:

What are the origins of Software Visualization?

How did it start?

How did it evolve?

What is the current status of the area?

Has the area evolved in the correct direction?

How fast has the area evolved?

Has the area adopted new technology?

Has the area evolved based on a well defined and unambiguous taxonomy?

Are there any misconceptions in the terminology used?

Has the area incorporated in its terminology the evolution of the area and the technology?

What approaches for visualization has been implemented so far?

Do they visualize the various software artifacts?

What are their key ideas and weaknesses?

What other, not yet implemented, proposals exist?

What are the challenges that SV is facing?

What can be done to boost the penetration of the area in the world of professionals?

What should be the characteristics of visualization tools in order to be easily accepted?

What framework should we establish to address these challenges?

D. METHODOLOGY

We will start by first exploring the area's historical roots and terminology. We will present the existing definitions for the most important terms and reveal any inconsistencies that exist. We will present and analyze the existing taxonomies of the area and reveal the "black spots" that have been unexplored, in the theoretical taxonomical level. Then we will examine a number of existing tools and analyze the principles that they are based on. After having a good insight into the current status, we will present the challenges that the area is facing and try to define a direction that should be followed for the future.

E. THESIS ORGANIZATION

Chapter II provides an overview of the area, its relation to Software Engineering, its importance and its potential ability to assist in the difficult task of

engineering the software. In this chapter, we also provide a brief history of the early years of the field and define the most common terms of the field.

In Chapter III we present all of the taxonomies and we analyze and compare them in an attempt to reveal their weak points based on the given evolution of software.

In Chapter IV we present Software Visualization tools that are significant, either from a historical point of view through their contribution of new principles or visualization metaphors, or by virtue of their acceptance by Software Engineering professionals.

Chapter V contains an overview of the challenges that should be addressed by researchers in the near or far future, if we expect Software Visualization to gain the role it deserves.

Finally, in Chapter VI we propose a new framework that can also be used as a taxonomy of the area or a vehicle for the development of new tools.

II. SOFTWARE VISUALIZATION OVERVIEW

A. INTRODUCTION

Computer technology has saturated the modern world, influencing every aspect of our lives. Technology is all around us: smart cars, smart houses, microwaves, wristwatches, factory automated installations, nuclear power plant control systems, aircraft, the personal computers we all use, Playstations and entertainment systems, to name a few.

Our lives are dependent on computer systems, to a degree that many of us have not yet realized. When we are flying in an aircraft, a computer is responsible for our safety by checking and controlling all the vital devices of the aircraft, reporting, of course, to the pilot. The nuclear plants are there providing energy without any problem because state of the art computer controlled systems are inspecting every small detail of their operation. The devices installed in hospitals providing oxygen to our loved ones, are providing the proper amount of oxygen based on a plethora of measurements taken every millisecond by various sensors that give feedback to their computerized control unit. When we press the brake pedal in our car, trying to avoid an obstacle, a signal is transmitted to a computer that directs the behavior of the braking system in a manner such that the minimum breaking distance is accomplished. Even the microwave we have in our house is preparing our food or turning on its internal light every time we open the door, under control of a computerized central unit.

Most important is that any computer based device would be useless without the existence of a piece of software inside it. All this great functionality that is hidden inside so many devices that we all enjoy, exists because someone designed and created the software for it.

But this is not all. Our requirements do not stop here. We want technology to be reliable since we are used to the service that it provides us. We want

technology to be safe, since we won't tolerate accidents due to a bug in the device. We want technology to be secure, because we care about our privacy but, on the other hand, we want technology to be easily accessible despite our own mobility or the limitations we pose regarding size, weight, price, speed, etc. Moreover, we want our devices to be upgradeable without having to pay a lot; we want them to be able to satisfy our continuously changing and sometimes poorly defined needs.

These are some of the reasons that made development of software in our era an extremely difficult and demanding activity. The final recipients of all those contradicting facts are the software engineers and programmers. These are the people who have to address those challenges and think of solutions, not in a theoretical space, but capable of being easily and inexpensively implemented.

B. SOFTWARE VISUALIZATION AND ITS RELATION TO SOFTWARE ENGINEERING

Software engineers must balance between user needs and constraints imposed by management. They have to do the job right within the given time limits. Software Engineering as a discipline, must address these challenges through the development and refinement of new models, techniques, practices, and tools that build upon sound engineering principles. A software engineering team must think of software not only as a mathematical description or a product, but also as a service, a commodity, or even as a user experience.

Among the roots of the various problems, the most significant has been widely accepted to be the nature of software. software engineers and programmers have to deal with an abstract entity that does not follow any of the laws of nature. Software is clearly abstract, dynamic, and extremely complex posing a lot of difficulties in its comprehension. Software is hidden from the senses, exists in its own space, is imaginary and can be perceived only using our imagination. Furthermore, analogies that apply in real life cannot be applied in

software. Making a scaled prototype of a ship is enough to extract critical facts regarding its behavior when later in the water. On the other hand, creating a database that can handle 10 users will not provide any information regarding its behavior if the number of users will be increased to 1,000,000. The problem of "paging yourself to death" is a well known problem for those that are aware of the history of Computer Science, showing clearly that scaling is not a solution for our field; at least not to the degree that it is for other science and engineering areas.

As we have already stated, the field of software engineering concerns itself with the technology and processes of software development, and thus it has approached the problems of software complexity and incomprehensibility in a number of ways. CASE (computer-aided software engineering) tools such as visual tools, are always welcome, as long as they have a smooth learning curve and can prove that they are cost efficient and deserving of the time that professionals will spend to embody them in their everyday practice.

Software visualization can be a significant aid for the perception and conceptualization of software and for clarifying software's inherent complexity.

Software visualization (SV), in general, is concerned with the presentation of the various software artifacts in a way that will decrease the cognitive load of any participant in the difficult task of engineering the software.

C. IMPORTANCE OF VISUALIZATION

The role that visualization plays for human reasoning has been expressed by philosophers throughout the centuries. A typical example is Kant's [6] statement "The senses cannot think. The understanding cannot see. By their union only can knowledge be produced," that shows clearly the need to stimulate all the senses in our fight against complexity.

Another typical example for the value of visualization and how a huge amount of information can be presented in an effective way, is the “Map of bread” shown in Figure 1.

Figure 1 La France des pains [7]

Human’s visual system has been acknowledged for its ability to quickly absorb information; combined with other senses the results can be tremendous. Presentation of complex concepts through the use of pictures can clarify them, and stepping slowly through a process, presented in a pictorial form, will increase the level of understanding. Viewing changes in a set of data or the outcomes of

software when different data sets are input to it, gives a quicker grasp of what is going wrong. Visual representations of run-time data can provide invaluable information in many phases of the software life cycle. Visualized presentations of large volumes of information, in the proper format, can help the assimilation of the information contained and offer a quick overview, usually required for gaining the “big picture.”

Those are only some of the reason that visualization has become an important tool in the hands of computer scientists and all engineers. Visualization is heavily used in mechanical engineering, chemistry, physics and medicine. Computer scientists have developed sophisticated systems to produce visualizations for these disciplines and, as a consequence, visualization has become a discipline of computer science and a valuable tool for Software Engineering.

In a survey made by R. Koschke [8] based on researchers from software maintenance, re-engineering and reverse engineering, 40% found software visualization absolutely necessary for their work and another 42% found it important but not critical.

Another survey conducted by Bassil & Kellller [9] showed that among the benefits of Software Visualization tools are savings in time and increases in productivity and software quality as results of better software comprehension.

As a conclusion, the area of Software Visualization, as a young discipline, is striving to find its way down the difficult road of software understanding. Many researchers have realized that it can greatly help Software Engineering in complicated tasks. Knowing that, modern computing systems are characterized by their power graphics, we have the burden, as researchers, to take advantage of those abilities but also of the evolution in relevant areas such as cognitive psychology, and create tools that will decrease the cognitive load of their users.

D. HISTORY OF SOFTWARE VISUALIZATION

The concept of software visualization dates back to the early days of computing, when Goldstein and von Neumann [10], in 1947, presented the flowchart as a mean of facilitating program comprehension, especially useful for programming in Assembly. This is generally accepted as the first effort for static visualization of programs. The Nassi-Shneiderman diagrams [11], presented in 1973, were a refinement of them.

For both approaches, various systems that automated their construction have been proposed. For example, Haibt [12], in 1959, developed a system to draw flowcharts automatically with a Fortran or assembly program as an input. Knuth [13], in 1963, developed a system to integrate documentation with the source code and also to automatically generate flowcharts. Roy and St. Denis [14], in 1976, developed a system for automatically generating Nassi-Shneiderman diagrams from source code.

In the area of “prettyprinting”, an effort was made, in 1963, by Naur [15] based on Algol language. This was followed by other ideas like a PL/I statement reformatter presented in 1970 by Conrow and Smith [16]; an automatic formatting system for Pascal, presented in 1977, by Hueras & Ledgard’s [17]; Knuth’s proposal in 1984 [18], for literate programming and many others up to Baecker & Marcus’ [19] effort, in 1990, with their SEE program visualizer – a UNIX-based system for automated typesetting C programs, according to an elaborate style guide.

Regarding the area of dynamic visualization, the first widely accepted effort is Knowlton’s [20] (1966) list processing and manipulation with its L6 low level list processing language, visualizing data structures in a dynamic way. The next significant step, in 1981, was Baecker’s [21] film “Sorting Out Sorting”, a 30 minutes color and narrated film that uses animated computer graphics to manipulate nine sorting algorithms. Other efforts in this area include Baecker's

interactive debugger, presented in 1968 [22], which was able to produce static images of high-level language data structures; Booth's[23] short film animating PQ-tree data structure algorithms, presented in 1975; Lieberman's [24] effort in 1984, to aid debugging of Lisp programs; up to the evolution of BALSA in 1984 by Brown and Sedgewick [25], an interactive tool for visualizing data structures in Pascal programs.

Some of the above mentioned SV tools will be presented later on, along with tools presented after 1990. For a more detailed history of SV, the reader is encouraged to read a paper written by R. Baecker & B. Price [26].

E. DEFINITIONS

1. Introduction

Communication requires commonly accepted definitions. Additionally, we believe that heedful, etymological analysis will reveal meanings not easily observable otherwise. Our effort is not to "reinvent the wheel" and provide our own definitions, but rather to present the existing ones, compare them, point out the "missing parts" and correlate them to the dimensions they offer to the area widely known as Software Visualization.

From the early beginnings of our exploration in this area, we realized that even the term Software Visualization itself is not well established and accepted. There are many students and researchers that prefer the "older" term of Program Visualization or even confuse Visual Programming with Program Visualization.

In our effort to define the boundaries of Software Visualization, we will start from the very beginning: the distinction between program and software; the differences between visualization, auralization, understanding, and conceptualization; the differences between software, software system, and programming, in general; and many more terms that we believe have caused the area to be in a continuous effort to show its significance in Software Engineering.

Although these may seem like philosophical questions, we believe that they will assist us in our effort to illuminate the existing dark spots in this area and point out a new direction, which we think is a promising one for the future of Software Visualization.

It is a well known fact that there are misunderstandings and confusions regarding the various terms used in the area of Software Visualization, partly due to the overlapping with other research areas and partly due to the fast evolution of the software itself, with the most distinctive example being the interchange of the terms program and software.

Of course it's out of the scope of this document to present all the existing definitions for those terms but only some of the most expressive and complete ones.

2. Literature Survey

a. Computer Program – Code

Etymologically, the word program is derived from the Greek language where the word means a sequence of statements that are pre-written or pre-defined that will be used later for execution.

According to Myers a program is "a set of statements that can be submitted as a unit to some computer system and used to direct the behavior of that system [27] & [28]." The interesting fact is that Myers never used the term software in the papers where he proposed his taxonomy.

In his classical book "The Mythical Man Month," Brooks present an evolution cycle, clearly separating the terms "program," "programming product," "programming system" and "programming systems product." According to this cycle, "a program is complete in itself," and will eventually "become a part of a collection of interacting programs, coordinated in function so that the assemblance constitutes an entire facility for large tasks [29]."

If we move to the legal system, we will see that "a computer program" is a set of statements or instructions to be used directly or indirectly in a

computer in order to bring about a certain result [30]" while the word coding, is defined as "transforming of logic and data from design specifications into a programming language [30]." Those definitions make the words coding and programming synonyms and, as a consequence, the words code and program.

Searching the IEEE standards to find the same definition we see that the term coding has two meanings, depending on the context: the first is exactly the same as the definition used in the legal system, while the second is more software engineering oriented. More specifically, coding is defined as:

> (1) In software engineering, the process of expressing a computer program in a programming language, and
>
> (2) The transforming of logic and data from design specifications (design descriptions) into a programming language [31].

As a necessary complement to the above definition we have to provide the definition of computer program, given in the same standard: "Computer program: A combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions [31]."

Two more interesting definitions are provided in the same standard, [31]; the first is for *source program* which is defined as "A computer program that must be compiled, assembled, or otherwise translated in order to be executed by a computer" and *code* which is defined as follows:

> (1) In software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or other translator. See also: source code; object code; machine code; microcode.
>
> (2) To express a computer program in a programming language.
>
> (3) A character or bit pattern that is assigned a particular meaning; for example, a status code.

IEEE recognizes different kinds of code, such as source code, machine code, compiler code, etc. with the most interesting definitions being the ones provided for *machine code* and *source code*. The first is defined as “Computer instructions and data definitions expressed in a form that can be recognized by the processing unit of a computer,” and the latter as “Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator. Note: A source program is made up of source code.”

Before we propose any definition, we believe that it’s worth mentioning two more facts regarding the IEEE’s view for the term program:

(1) IEEE has depreciated terms such as computer program component and computer program configuration item by transforming them to computer software component and computer software configuration item; a transformation indicating the significant difference between the two terms.

In their various definitions for program and code, they intentionally repeat the phrase “data definitions” separating the actual data that will be used as input during the execution of the program from the program itself.

IEC defines program as “A series of actions proposed in order to achieve a certain result [32].” Musa et al. define program as “A set of complete instructions (operators with operands specified) that executes within a single computer and relates to the accomplishment of some major function [33].”

Finally, Smith & Wood define programs as: “A set of coded instructions which enable a computer to function. A program may consist of many modules and be written in assembly or high-level language. Note the spelling "program", whereas "programme" is used to describe a schedule of tasks [34].”

This survey of definitions ought to close by citing the definition provided by Turing in one of his papers:

> If one wants to make a machine mimic the behaviour of the human computer in some complex operation one has to ask him how it is done, and then translate the answer into the form of an instruction table. Constructing instruction tables is usually described as "programming." To "programme" a machine to carry out the operation A, means to put the appropriate instruction table into the machine so that it will do A [35].

b. Software

According to the literature, software is a more generic term. The term was first used (in the area of Computer Science) by John Wilder Tukey [36] in the phrase "Today the ‘software' comprising the carefully planned interpretive routines, compilers and other aspects of automative programming are at least as important to the modern electronic calculator as its 'hardware' of tubes, transistors, wires, tapes and the like."

According to the American Heritage Dictionary, software is “The programs, routines, and symbolic languages that control the functioning of the hardware and direct its operation [37].”

In the literature, software is usually used as a generic term for collections of computer data and instructions or even anything that can be stored electronically, being the complement of hardware in terms of computers. The former requires the existence of the latter while the latter without the former is useless. The term “soft” contained in the former is often used to denote its modifiable nature in contrast with the concrete and unchanged nature of the latter.

Pressman, in one of his classic books, defines software as: “(1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs [38],” making clear that software is a more generic term than program.

IEEE defines software as “computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system [31],” also pointing out that software is a more generic term than program.

Definitions, identical or similar to that of IEEE, can be found in UK MoD publications [39], ESA publications [40] and in the majority of governmental publications.

In general, the term software is defined as a more generic term usually including the term program but, occasionally, used interchangeably with it.

c. Algorithm

Here things are more clear since the various definitions that can be found in the literature are almost identical.

In their classical book on algorithms, used by the majority of the Computer Science departments worldwide, T. Cormen et al. define the term algorithm as “any well defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. We can also view an algorithm as a tool for solving a well-specified *computational problem* [41].”

In another book where algorithms are presented in the context of Computer Science, an algorithm is defined as a “a specification of behavioural process…. consists of a finite set of instructions that govern behaviour, step by step [42],” while Price, Baecker and Small describe algorithms differently as “higher-level descriptions of programs [43].”

IEEE defines the term algorithm as “A finite set of well-defined rules for the solution of a problem in a finite number of steps [44],” while the Oxford Dictionary states that an algorithm is a “process or set of rules used in calculations or other problem-solving operations [45].”

We have to point out that definitions of algorithm (in the context of computer science) present one major difference from a theoretical perspective; whether the number of steps that have to be executed should be finite or not.

d. Visualization

In order to define terms such as Software Visualization we have to look at both parts individually but also in conjunction, since the meaning may be altered when they are joined.

The Oxford Dictionary [46] defines the word visual in seven different ways: six closely related to sight and the seventh one as “Of the nature of a mental vision; produced or occurring as a picture in the mind,” while the word visualization is clearly defined as “(1) The action or fact of visualizing; the power or process of forming a mental picture or vision of something not actually present to the sight; a picture thus formed, (2) The action or process of rendering visible.” It is also worth mentioning that the term visualize is defined as

> (1)To form a mental vision, image, or picture of (something not visible or present to the sight, or of an abstraction); to make visible to the mind or imagination, (2) To form a mental picture of something not visible or present, or of an abstract thing, etc.; to construct a visual image or images in the mind, (3) to render visible.”

Searching other dictionaries we found that the creation of a mental image is mentioned in the definition of the word visual, as one of the possible meanings, but not the primary one.

> In [47] the word visual is defined as
>
> (1) of or relating to the sense of sight: a visual organ; visual receptors on the retina. (2) Seen or able to be seen by the eye; visible: a visual presentation; a design with a dramatic visual effect. (3) Optical. (4) Done, maintained, or executed by sight only: visual

navigation. (5) Having the nature of, or producing an image in the mind: a visual memory of the scene. (6) Of or relating to a method of instruction involving sight.

In [48] the term visual is defined as "(1) of, relating to, or used in vision, (2) attained or maintained by sight, (3) VISIBLE, (4) producing mental images, (5) done or executed by sight only, (6) of, relating to, or employing visual aids."

Finally, McCormick, De Fanti & Brown in [49] define visualization as "the use of computers or techniques for comprehending data or to extract knowledge from the results of simulations, computations, or measurements."

e. Software Visualization

According to some of the pioneers in the area of Software Visualization, the term visualization takes its meaning from the idea declared by the seventh definition of the term, referring to an older version of the Oxford Dictionary, where visualization was defined as "the power or process of forming a mental picture or vision of something not actually present to the sight [43 page 3]. Software Visualization is actually defined the same way it was defined in [50][1] as "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software." According to our understanding, this definition is not really broad since the means that can be used in the "visualization" process should be among the enumerated ones in the definition and hence not all different sensory inputs are allowed. A consequence of this definition of SV is that it's subject to future inventions and thus it will have to be altered if new technological inventions will be used.

[1] This was expected since the authors of Chapter 1 of [43] are the ones that published [50] some years earlier.

Roman and Cox define SV as “a mapping from program to graphical representations [51].” Again here we see the limitation of SV in graphical means with the object of visualization being only a program.

Koschke, in his well-known survey on Software Visualization [8] mentions that “Software visualization is concerned with the static visualization as well as the animation of algorithms, programs, and the data they manipulate.”

More recent views of Software Visualization broaden the area. A typical example is the description provided by Diehl in [52] where it is stated that “Software visualization encompasses the development and evaluation of methods for graphically representing different aspects of software, including its structure, its execution, and its evolution.”

f. Program Visualization

Here things become a little more complex, since the majority of the published papers agree that program visualization is a branch of Software Visualization that is focused on programs, compared to the other part of Software Visualization that concerns algorithms.

For example, in [50] program visualization is defined by Price, Baecker and Small as “the visualization of an actual program code or data structured in either static or dynamic form.”

The term “program visualization” became widely accepted after Myers’ papers [27] [28] and has been expanded to what we call today Software Visualization after the incorporation of the concept of algorithm animation.

As we will see later in Myers taxonomy presentation, he does not provide an explicit definition but rather a descriptive one, stating that “in Program Visualization the program is specified in the conventional, textual manner, and the graphics are used to illustrate some aspect of the program or its run-time behavior [27] [28].”

Roman and Cox, define program visualization as “a mapping from program to graphical representations [51],” which is a very broad definition but

clearly does not contain the concept of algorithm animation.[2] Oudshoorn et al. define PV as “the use of graphical artifacts to represent both the static and dynamic aspects of a program [53].”

Ellershaw & Oudshoorn [54] wrote in a technical report, “Program visualization focuses on the graphical representation of an executing program and its data. The information is presented in a form designed to enhance both the understanding and productivity of the programmer through the efficient use of the human visual system,” and later defined as “the application of graphical transformations to an executing program to enhance the reader’s understanding of that program.”

Baecker, one of the founding fathers of visualization defines program visualization as "the use of graphics to enhance the art of program presentation and thereby to facilitate the visualization, understanding, and effective use of computer programs by people [55].” Petre et al. describes the purpose of program visualization as “trying to find simplicity in a complex artifact (e.g. thousand-line code), to produce a selective representation of a complex abstraction [56].“

In general, according to the majority of the papers related to this area, program visualization is directly connected to the term “program” and the concept of assisting the user to better understand it. We also have to mention that in the early days of the discipline, it was often confused with software visualization and both terms were used interchangeably. After the taxonomy proposed from Price et al. [50], and the publication of [43], it became widely accepted that program visualization is a part of software visualization but still, not all software artifacts were included in the definition.

g. Algorithm Animation – Algorithm Visualization

For many researchers the words animation and visualization seem to be identical and hence they are used interchangeably. In one of the early papers for algorithm animation, Brown states that “algorithm animation is a form

[2] This can be easily seen by the taxonomy they proposed.

of program visualization… Algorithm animation displays are, essentially, dynamic displays showing a program's fundamental operations – not merely its data or its code [57]."

Price, Baecker and Small [43] distinguish between algorithm animation and algorithm visualization stating that "algorithm visualization is the visualization of the higher level abstractions which describe software," while "algorithm animation is dynamic algorithm visualization."[3] They divide the area this way, into static algorithm visualization and algorithm animation, providing as an example of the former the well-known flowcharts and as examples of the latter tools like Balsa, Zeus, Tango etc.

Karren & Stasko [58] state that an algorithm animation "visualizes the behavior of an algorithm by producing an abstraction of both the data and the operations of the algorithm," thereby loosening the distinction made by Price, Baecker and Small.

h. Static – Dynamic Visualization

Etymologically, these word are derived from Greek, with the word "static" meaning – among other things not relevant to our area – something that does not change, that remains stable and in the same state, and the word "dynamic" meaning something that changes over time, that is active, energetic, effective, and forceful and in our context is the opposite of static. In general, these terms refer to systems that can show "snapshots" of the object to be visualized or systems that can provide a "live" representation, respectively.

i. Visual Programming

Even though almost all the papers of the area state that there is a misunderstanding between Visual Programming and Program Visualization, all the authors provide similar definitions that do not actually allow for distinctions. Myers [28] states that

[3] Price, Baecker, and Small define both areas of study to collectively be a part of Software Visualization.

> Visual Programming as 'Visual Programming' (VP) refers to any system that allows the user to specify a program in a two-(or more)-dimensional fashion. Although this is a very broad definition, conventional textual languages are not considered two-dimensional since the compilers or interpreters process them as long, one-dimensional streams.

Price, Baecker and Small in [43] define Visual Programming as “the use of ‘visual’ techniques to specify a program.”

Searching for definitions from Visual Programming researchers, we found that there exist definitions that clearly separate the two areas, like the one provided by Burnett [59] who states that

> Visual programming is programming in which more than one dimension is used to convey semantics. Each potentially significant multi-dimensional object or relationship is a token (just as in traditional textual programming languages each word is a token) and the collection of one or more such tokens is a visual expression such as diagrams, free-hand sketches, icons and so on. When a programming language’s (semantically significant) syntax includes visual expressions, the programming language is a visual programming language (VPL).

Contrastingly to this, there exist a number of other definitions that actually merge the two areas. For example, Golin and Reiss [60] state that

> A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions. The latter is taken to be the definition of a visual programming language… Visual programming environments provide graphical or iconic elements, which can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction.

McIntyre and Glinert [61] define Visual Programming as

> the use of visual expressions (such as graphics, drawings, animation or icons) in the process of programming. These visual expressions may be used in programming environments as graphical interfaces for textual programming languages; they may be used to form the syntax of new visual programming languages

> leading to new paradigms such as programming by demonstration; or they may be used in graphical presentations of the behavior or structure of a program.

As a conclusion we have to point out that in all definitions provided by researchers in the area of Software Visualization, the common thing is the concept that Visual Programming aims in the specification of a program in means different than the traditional textual one and also that the traditional textual format is not considered to be part of Visual Programming while things in the arena of Visual Programming are not the same.

j. Taxonomy

Etymologically, the word is derived from Greek and more specifically, as listed in the Oxford Dictionary [46] from the words τάξις (taxis) that means arrangement or order, and -νομία (nomia) that means distribution. English words that are usually used as synonyms to taxonomy are “classification” and “categorization.” One of the major disputes relating to the word taxonomy is whether it contains the concepts of mutual exclusiveness and joint exhaustiveness for the categories it specifies.

Aristotle in one of his earlier works, the “Categories,” proposed one of the first taxonomies but he never mentioned anything regarding mutual exclusiveness [62]. Wikipedia [63] states that

> Taxonomies are frequently hierarchical in structure, having parent child relationships. However, taxonomy may also refer to relationship schemes other than hierarchies, such as network structures. Other taxonomies may include single children with multi-parents, for example, "Car" might appear with both parents "Vehicle" and "Steel Mechanisms"; to some however, this merely means that 'car' is part of several different taxonomies… Mathematically, a hierarchical taxonomy is a tree structure of classifications for a given set of objects. At the top of this structure is a single classification, the root node that applies to all objects. Nodes below this root are more specific classifications that apply to subsets of the total set of classified objects.

According to IEEE [64] taxonomy is defined as “a scheme that partitions a body of knowledge and defines the relationships among the pieces. It is used for classifying and understanding the body of knowledge.”

Webster’s Dictionary, defines taxonomy as “A systematic arrangement of objects or concepts showing the relations between them, especially one including a hierarchical arrangement of types in which categories of objects are classified as subtypes of more abstract categories, starting from one or a small number of top categories, and descending to more specific types through an arbitrary number of levels [48].”

One of the most cited and broadly used definitions is the one provide by Whatis.com that defines taxonomy as:

> the science of classification according to a pre-determined system, with the resulting catalog used to provide a conceptual framework for discussion, analysis, or information retrieval. In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that are mutually exclusive, unambiguous, and taken together, include all possibilities [0].

III. SOFTWARE VISUALIZATION TAXONOMIES

A. IMPORTANCE OF TAXONOMY

As an old Chinese proverb says “The first step towards wisdom is calling things by their right names,” so the creation of a taxonomy is the first step towards the exploration of an area.

A properly defined taxonomy will provide a framework for discussion, analysis and research guidance by offering a systematic and systemic overview of the area. Structuring a research area and defining its boundaries provide a better understanding, both for students and researchers. This is required because it’s not unusual to see “area gurus” be biased in their opinion due to their personal understanding and definition, while on the other side, new researchers feel lost and wander around without the “big picture” being offered to them.

A well-founded taxonomy, with the proper characteristics, can further investigation in any field of study. A common language or terminology and the existence of a roadmap for a research area promote communication of new ideas and allow new discoveries to be identified and explored. A well determined taxonomy might also reveal where a probable new discovery, or a variation of an existing idea, can be found, since unexplored areas can be more easily identified.

We strongly believe, that the existence of a solid, comprehensive scheme for classifying existing principles and recommended practices in conjunction with the existence of a framework that will not alter as technology evolves, is a basis that will guide researchers into new exploration of the area and as such, a taxonomy based on criteria that are loosely connected to specific technologies and practices but tight with principles, is required.

In order to accomplish this, we will present and analyze the existing taxonomies and the various definitions that exist in our research area.

B. EXISTING SOFTWARE VISUALIZATION TAXONOMIES

Throughout the years, various researchers have proposed a number of taxonomies, mainly for the classification of the Software Visualization tools or the domain of Software Visualization itself. These taxonomies will be presented in the following paragraphs.

1. Myers

Among the most well known and frequently referenced papers in the area are the two papers published by Brad A. Myers [27] [28], who realized early on, the increase in the interest in Software Visualization and the significant impact successful visualizations might have in the alleviation of software complexity.

In his first paper published in 1986, he proposed two taxonomies, one for "Program Visualization Systems" and one for "Visual Programming."

Myers, defines a program "as a set of statements that can be submitted as a unit to some computer system and used to direct the behavior of that system," a definition that can also be found in [66].

Regarding the term program visualization, Myers does not provide an explicit definition but rather a descriptive one, stating that "in Program Visualization the program is specified in the conventional, textual manner, and the graphics are used to illustrate some aspect of the program or its run-time behavior [27]."

Based on the above definition and the reasonable conclusion that PV systems accept as input a program made in the conventional textual manner, his taxonomy divides the area of Program Visualization using two axes, as shown in Table 1. The first axis answers the question "*What is to be visualized?*" The answer is influenced by the classical definition of the word "program" and hence

there are two sub areas defined, code and data. The second axis partially answers the question "*How is it to be visualized?*" This divides the area into static and dynamic visualizations not taking into consideration all the other issues such as the modalities, the interactivity, the medium, etc.

According to this taxonomy, systems that will attempt to visualize a program will choose to visualize either the code or the data in a dynamic or static way or even a combination of those.

	Static	**Dynamic**
CODE	Flowcharts, SEE Visual, Compiler, PegaSys	BALSA, PV Prototype
DATA	TX2 Display Files, Incense	Two Systems, Sorting out Sorting, BALSA, Animation Kit, PV Prototype

Table 1 Classification of Program Visualization Systems, as initially proposed by Myers [27]. In this taxonomy PV systems are classified by whether they illustrate code or data, and whether the produced visualizations are static or dynamic.

Four years later, Myers published an updated version of his taxonomy [28] by expanding the term "program" to include the concept of "algorithm," even if the initial definition of the term program was not altered. The new taxonomy has the third option of "algorithm visualization" across the first axis and is shown in Table 2.

As a consequence of this addition and assuming that an algorithm is actually a higher level of abstraction of a program, the term "software" is more closely approached with this taxonomy.

	Static	Dynamic
CODE	Flowcharts, SEE Visual, Compiler, PegaSys, TPM	BALSA, PV Prototype, MacGnome, Object Oriented Diagram, TPM
DATA	TX2 Display Files, Incense	Two Systems, Sorting out Sorting, BALSA, Animation Kit, PV Prototype, ALADDIN, Animation by Demonstration, TANGO
ALGORITHM	Stills	Two Systems, Sorting out Sorting, BALSA, Animation Kit, PV Prototype, ALADDIN, Animation by demonstration, TANGO

Table 2 The updated classification of Program Visualization Systems proposed by Myers [28] classifying programs by whether they illustrate code, data or algorithm, and whether the produced visualizations are static or dynamic.

2. Price et al. (1992)

A couple of years after Myers published his updated taxonomy, a new taxonomy was proposed based on the observation that previously proposed taxonomies[4] "use few dimensions and do not span the space of important distinction between systems [67]."

The main purpose of this proposed taxonomy, which is based on the same definition as [27] & [28], was to provide a framework for the evaluation of Software Visualization systems, capable of providing a clear picture of the

[4] The authors mention two taxonomies: The one proposed in [27] & [28] by Myers and a taxonomy that has been proposed in [68]. The first taxonomy has already been presented while the latter is not a taxonomy but rather a list of attributes, required for an effective SV tool.

features and abilities of a specific system or of distinguishing the differences among systems we would like to compare.

The basic concept behind this new taxonomy is the introduction of a number of characteristics for evaluating a Software Visualization system (in this case thirty such characteristics), which will cover the space of available features and capabilities and hence classify the available tools.

The new taxonomy initially groups the characteristics of a Software Visualization system, initially in six main categories and then subdivides each group of characteristics into subcategories, and is shown in Table 3. Thus the taxonomy can be viewed as an expandable n-ary tree.

Using this taxonomy, a Software Visualization system can be labeled based on its properties for each of the subcategories produced.

According to the authors:

- The **Scope** category describes some general attributes of a system
- The **Content** category answers the questions "*What is to be visualized?*"
- The **Form** category answers the question "*What are the visualization elements?*"
- The **Method** category answers the question "*How is the visualization specified?*"
- The **Interaction** category answers the question "*How do we interact with and control the visualization*?"
- The **Effectiveness** category answers the question "*How good is the visualization?*"

TOP LEVEL	Scope	Content	Form	Method	Interaction	Effectiveness
SECONDARY	System/ Example	Program/ Algorithm Visualization	Medium	Specification Style	Navigation	Appropriateness & Clarity
	Class of Programs	Code Visualization	Graphical Elements	Batch/Live	Elision	Experimental Evaluation
	Scalability	Data Visualization	Color	Fixed / Customizable	Temporal Control Mapping	Production Use
	Multiple Programs	Compile/ Run-Time	Animation	Code Familiarity		
	Concurrency	Fidelity and Completeness	Multiple Views	Invasive		
	Benign / Disruptive		Other Modalities	Customization Language		
				Same language		

Table 3 The Price et al. (1992) taxonomy at a glance

3. Roman and Cox

The same year a new taxonomy was proposed that is based on a more "relaxed" definition of program visualization that tried to offer a new perspective in the term. According to the authors [51], program visualization is defined as "a mapping from program to graphical representations," and as such the proposed taxonomy takes into consideration the participants in the visualization process and also the domain, the range and the nature of the mapping itself.

The proposed taxonomy is shown in Figure 2. More specifically, they distinguish three participants[5]

[5] It is important to mention the authors' comments on the three roles of the proposed taxonomy: "While, these are only stylized roles meant to help us organize and present the material, the specialized expertise required of each role may actually lead in practice to a return division of labor among distinct individuals [51]."

- The **programmer** who develop the program,
- The **animator** who defines and constructs the mapping, and
- The **viewer** who observes the graphical representation.

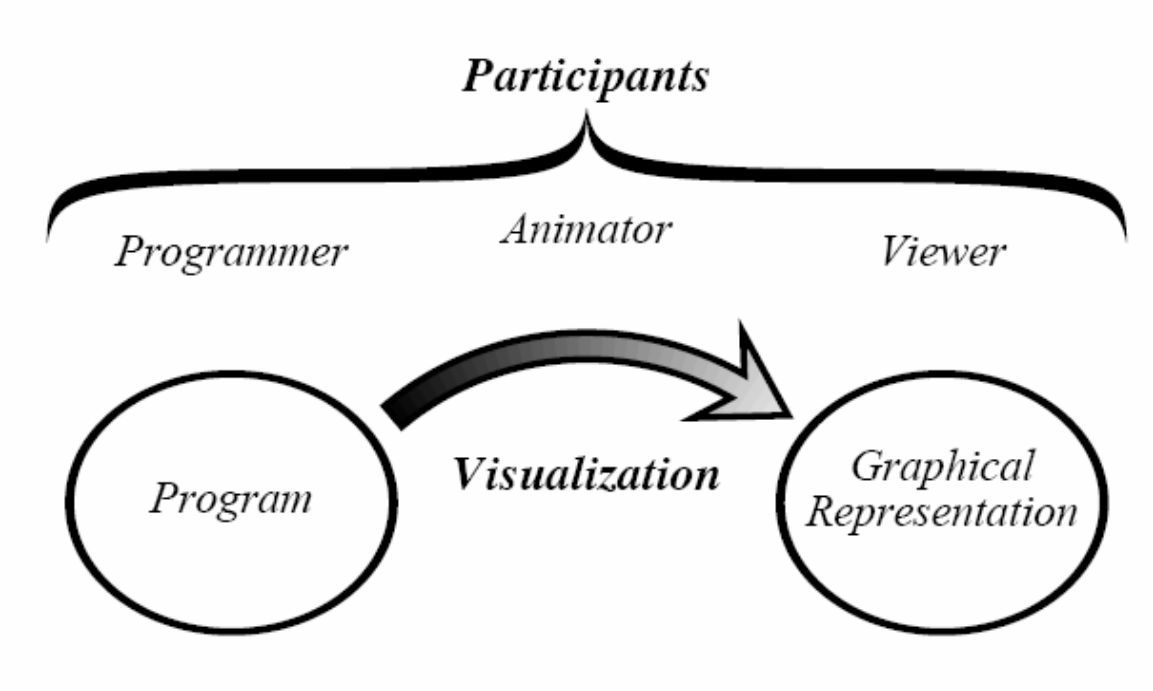


Figure 2 Taxonomic criteria and roles in program visualization as proposed by Roman & Cox [51]

The taxonomic criteria are presented in three axes:

- **Scope**: This category of criteria answers the question “What is to be visualized?” adding some interesting concepts in the answer, compared to the preceding taxonomies, such as the program’s control states and its execution behavior.
- **Abstraction**: This term is introduced for the first time in the area (even if the concept is mentioned in [69] & [57] with the terms

"sophistication degree" and "content" respectively). According to the authors, here someone can find the answer to the question "What kind of information is conveyed by the visualization?**"**

- **Specification Method**: The criteria included in this category, according to the authors, answer the question "How is the visualization constructed?**"** addressing things concerning the level of interactivity and some technical issues regarding the creation of the visualization itself (e.g. event or state transition oriented, need modification of code, etc.)
- **Technique**: This area, according to the authors, deals with the answer to the question "How is the graphical representation used to convey the information?**"** mainly addressing effectiveness issues of the visual communication.

Finally, we have to mention that for every category of taxonomic criteria there are a number of sub-criteria that may be used to classify a system. The full list of the proposed criteria is shown in Table 4.

TOP-LEVEL	Scope	Abstraction	Specification Method	Technique
SECONDARY	Code	Direct representation	Predefinition	Sample Execution selection
	Data State	Structural Representation	Annotation	Screen Design
	Control State	Synthesized Representation	Association	Information Encoding
	Behavior	Analytical Representation	Declaration	Presentation Enhancements
			Manipulation	

Table 4 Roman & Cox taxonomy at a glance

4. Stasco and Patterson

The same year, another taxonomy for software visualization systems was proposed by Stasco and Patterson [70]. It follows the concept presented by Myers [28] but expands it into a four-dimensional space. The concept of labeling Software Visualization systems is being rejected and they promote the idea of showing "how different systems exhibit varying levels of the four dimensions [70]."

More specifically the axes they propose to span the area of software visualization are:

- **Aspect**: This axis is similar to the first axis presented by Myers [28] and almost the same as the "Scope" category of [51].[6] This axis answers the question "What is to be visualized?" The difference is that under the concept of "aspect" the authors include all the different views that can be visualized; from simple textual views or diagrammatic ones (such as Nassi-Schneiderman diagrams) to data structures, flow of control, clauses and goals (for logical programming environments), list structures and function calls (for functional languages), including also algorithm animation.

- **Abstractness**: This area deals with the level of abstraction that is used from the visualization tool to display the required information and as such, is similar to the second category of [51]. The authors mention the isomorphism of the displayed information to the components they represent; a concept that is also addressed by [51] under the term "Direct representation." They also introduce the concept of "intention content [70]" defined as the "semantics or meaning behind otherwise context-free data and code" which reveals the fact that in visualizations of higher intention content, the

[6] We have to mention that although Stasco's & Patterson's taxonomy chronologically is later than the taxonomy proposed by Roman & Cox, the latter is not mentioned by the authors (they differ by four months in their publication dates) and hence any common results should be interpreted as independently found.

programmer's purpose must be well defined and stated before the production of the visualization in order for the latter to be more effective.

- **Animation**: According to the authors, "this classification dimension describes the dynamics or animation shown in software visualization systems [70]." The authors intentionally avoid the use of the terms static and dynamic and they explore this area, first by providing their own definition for animation and second by introducing the notion[7] of valid configuration of a program as a state of the program that involves semantic meaning and that is reachable during execution. In other words, valid configurations provide the points during the execution of the algorithm that give meaning in terms of the program's purpose and functionality in contrast to program's valid states that may be reached between two different valid configurations. This classification category seems to partially answer the same question as the category "abstraction" of [51] but also partially the question of "specification method" of [51].
- **Automation**: This classification axis concerns required user intervention and effort for the creation of visualization pointing out also the fact that high levels of automation are not always possible.

5. Price et al. (1993)

One year after the publication of their initial taxonomy, Price et al. proposed, "a new taxonomy for systems involved in the visualization of computer software [50]," which is actually an enriched version of the previous one but this time characterized as "principled."

They first analyze some of the existing definitions and then provide their own, according to which "Software Visualization is the use of the crafts of

[7] The authors seem to introduce this term only for animated visualizations.

typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software."[8]

Based on this definition they provide a Venn diagram for the area of Software Visualization showing how the various terms fit together, as shown in Figure 3.

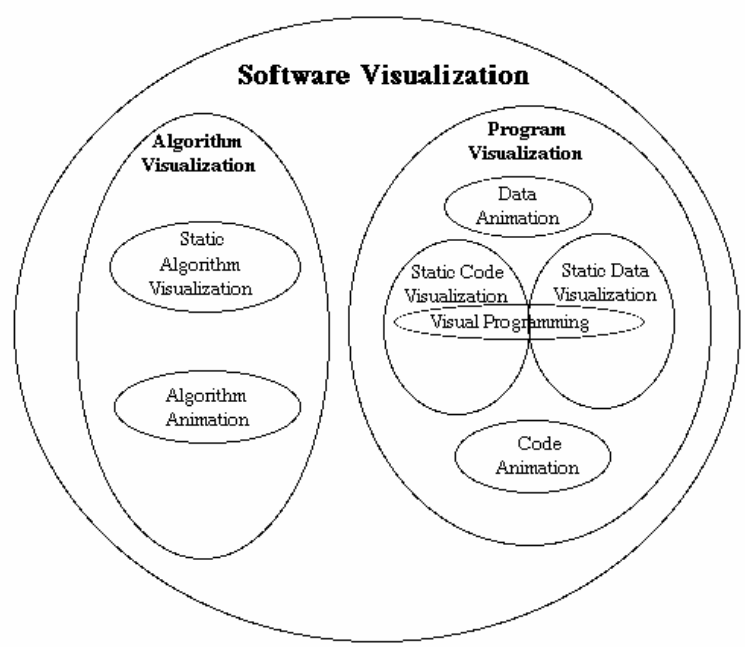


Figure 3 A Venn diagram showing how each of the existing (at that time) terms in the literature fit together[50].[9]

Compared to their previous taxonomy, a new concept they introduce is the various humans' roles that are involved in the process of visualization, dividing those roles in similar categories to Roman & Cox in [51]. More specifically they distinguish the following four roles:

[8] Their definition is based on the definition provided at [71] where Program Visualization is defined as "the use of the techniques of interactive graphics and the crafts of graphic design, typography, animation and cinematography to enhance the presentation and understanding of computer programs."

[9] The reader is encouraged to see the updated diagram published in [43]

- The **programmer** who wrote the original program,
- The **SV software developer** who wrote the system used to create the visualization,
- The **visualizer** who constructs the visualization, and
- The **user** who is the person using the visualization to understand the program, mentioning of course that a person can have more than one role.

They also emphasize the fact that visualizations produced by any system are tightly coupled to the visualization's final user's need for a mental picture.

Although the authors have analyzed the various roles users have in visualization, the changes they made in their taxonomy are not affected by this, but mainly concern the expansion, reorganization and redefinition of the taxonomy's categories, with the terms used for the primary categories unchanged, as shown in Figure 4.

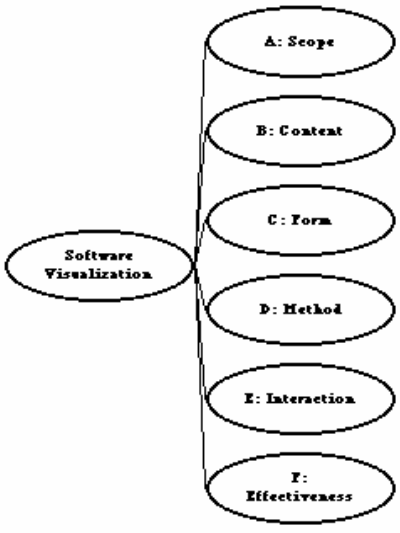


Figure 4 Primary categories of Price et al. updated Software Visualization [50].

According to the authors:

- The **Scope** category, in this new taxonomy, describes the "range of software that can be handled by a given Software Visualization system," and hence a new set of subcategories (as shown in Figure 5) has been introduced, differentiating this category significantly, compared to the older one.

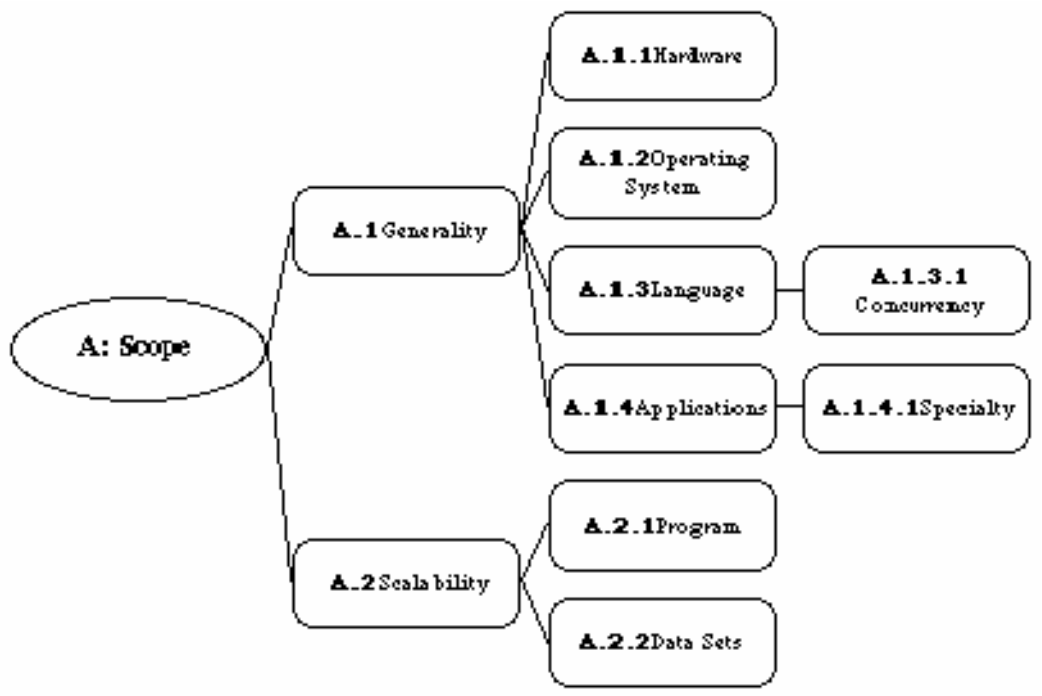


Figure 5 Sub-categories for the Scope category of Price et al. updated taxonomy [50]

- The **Content** category describes the subset of information that a particular Software Visualization system really uses in constructing the visualization, addresses time issues regarding the gathering of data required for the visualization construction and finally concepts related to the isomorphism of the presented visualization compared to the actual input. This category is based on the observation that

there is no tool that can visualize all of the information hidden inside software. One interesting point is that they differentiate program from algorithm in a user-centric way:

"if the system is designed to educate the user about a general algorithm, it falls into the class of algorithm visualization. If, however, the system is teaching the user about one particular implementation of an algorithm, it is more likely program visualization."

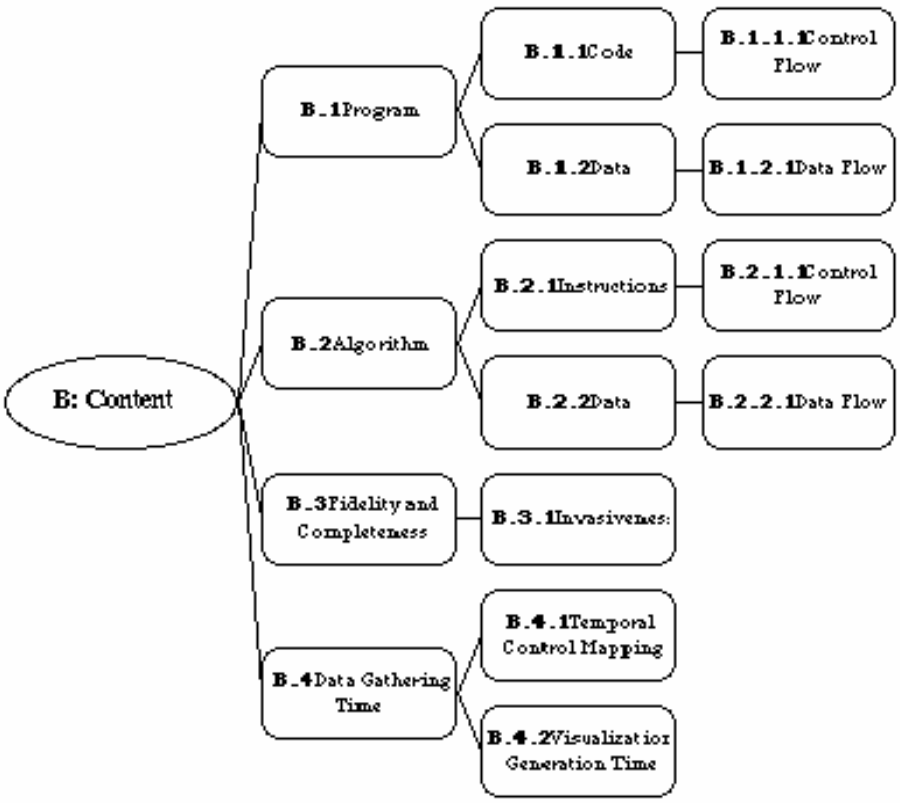


Figure 6 Sub-categories for the Content category of Price et al. updated taxonomy [50]

- The **Form** category describes the characteristics of the output of the visualization and classifies aspects such as the target medium, the graphical elements used to produce the visualization, the

granularity offered, the elision capabilities, the number of simultaneous views of different aspects offered, etc. The sub-categories of this category are shown in Figure 7.

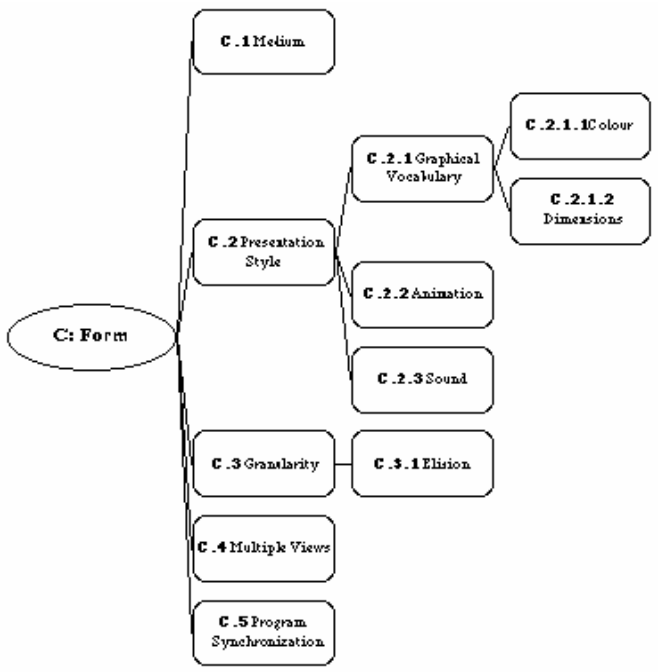


Figure 7 Sub-categories for the Form category of Price et al. updated taxonomy [50]

- The **Method** category concerns the way visualization is specified separating Software Visualization systems describing the style in which the visualizer specifies the visualization and one describing the way in which the visualization and the program source code are connected.

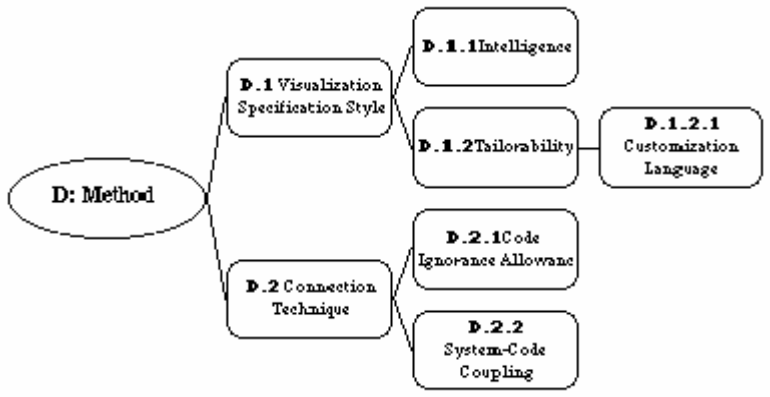


Figure 8 Sub-categories for the Method category of Price et al. updated taxonomy [50]

- The **Interaction** category seems to answer the same question as the one proposed in their previous paper but some of the sub-categories proposed are reorganized in a different way as shown in Figure 9.

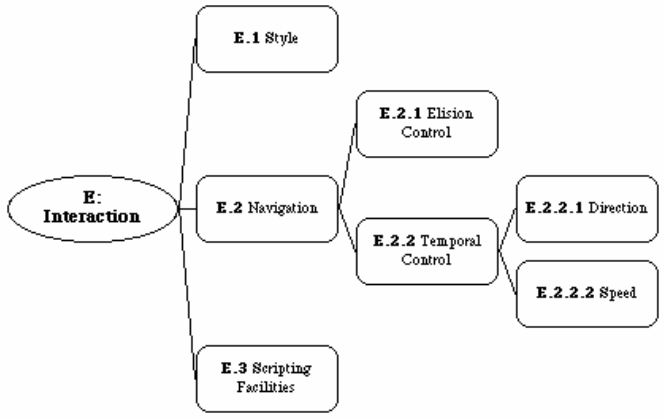


Figure 9 Sub-categories for the Interaction category of Price et al. updated taxonomy [50]

- The **Effectiveness** answers again a similar question to that of the previous taxonomy proposed by the authors but new sub-categories have been added and some older ones have been altered as shown in Figure 10.

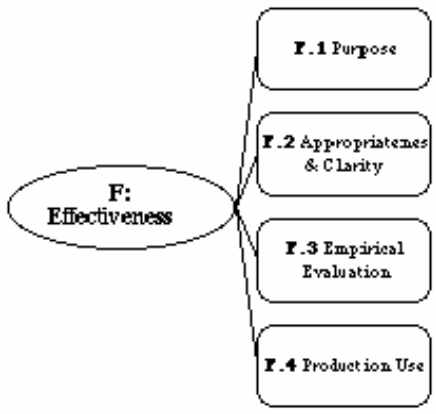


Figure 10 Sub-categories for the Effectiveness category of Price et al. updated taxonomy [50]

6. Oudshoorn et al.

The taxonomy proposed by Oudshoorn et al., follows a different approach than those previously presented. They accept the fact that the aim of any visualization is to assist the programmer in his mental model creation procedure regarding a piece of software and they propose a taxonomy “based on the notion of what a user can expect from program visualization, from the system point of view [53].”

They define PV as “the use of graphical artifacts to represent both the static and dynamic aspects of a program [53];” a definition based on [28] pointing out the relation of the PV to the program development life cycle.

They first present three aspects for program visualization as shown in Figure 11. The ”purposes” aspect concerns the use of the visualization that can be either for program understanding or debugging, or for performance analysis. The “mechanisms” aspect concerns the way visualization is produced and the “ideals” aspect concerns the required attributes for a useful visualization.

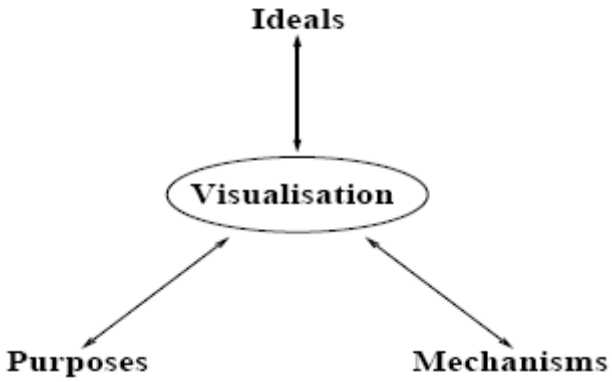


Figure 11 Program visualization aspects as proposed by Oudshoorn et al. [53]

Their proposed taxonomy is shown in Figure 12 and is actually centered in the entities that are to be visualized, also taking into consideration the different uses as a second axis attached to every element to be visualized.

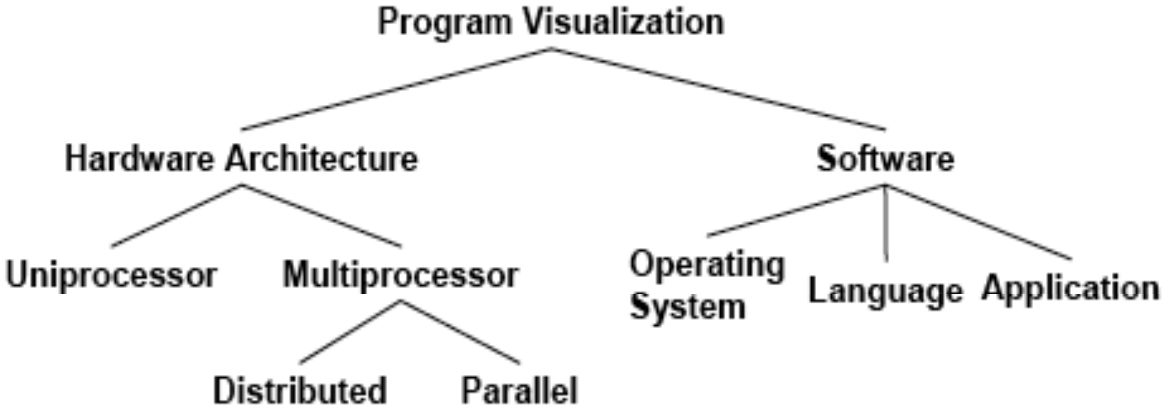


Figure 12 Taxonomy proposed by Oudshoorn et al. [53]

As a final note for this taxonomy, it is worth saying that, according to the authors, the program's abstract representations created should be differentiated

from the associated pictorial representations, meaning that "how" they are displayed is coupled with the ideals of visualization and not with the abstractions themselves.

7. Tilley & Huang

During a project on reuse of legacy systems, the authors of [72], realized that the majority of existing Software Visualization tools, even if presented as an effective means in the fight against software's increased complexity, have not proven that graphical representations are superior to the traditional textual ones.

In their effort to create a comparison framework for evaluating the capabilities of the various tools in the context of their industrial partner numerous constraints, they developed a descriptive classification of program visualization techniques with main axes determined by the level of interaction between the users and the generated graphical documentation.

Based on the fact that the maintenance of a legacy system requires the prior understanding of the software, they view program visualization as a re-documentation process.

They argue that a task-oriented classification of PV area will be more effective from the user's perspective since it will "map common activities related to program understanding… to specific types of software visualization ([72] page 227 emphasized)." They claim that no matter how the graphical views are produced, the real issue is that Software Visualization tools should address the everyday problems faced by people dealing with real large scale software projects, otherwise any effort will remain in the theoretical academic space.

They first define Software Visualization as "a special type of information visualization that uses computer graphics and animation to help illustrate and present computer programs, processes and algorithms [72];" a definition based on the various aspects of software.[10]

[10] The provided definition can be found at Georgia's Institute of Technology web site, at www.cc.gatech.edu/gvu/softviz, a department directed by Prof. J. Stasko. Last date accessed?

After providing a definition, they propose a task-oriented taxonomy, which is not strictly disjoint, since some tools may fall in more than one category. It is important to mention the authors' comments on the three roles of the proposed taxonomy: "While these are only stylized roles meant to help us organize and present the material, the specialized expertise required of each role may actually lead in practice to a return division of labor among distinct individuals" in more than one category.

More specifically, they categorize Software Visualization tools as follows:

- Tools that produce **static** visualizations: They claim that these tools are more suitable for high-level representation of overall software architecture, files and functions dependency, etc. especially when those views can be presented in different level of abstractions.
- Tools that produce **interactive** visualizations: They define as interactive visualizations the ones that allow the user to navigate the produced diagrams and claim that these kinds of tools, assuming they also provide the proper navigation facilities, provide significant aid in understanding the program's behavior.
- Tools that produce **editable** visualizations: According to the authors, editable visualizations are those that allow the user to make changes to the generated graph. The authors argue that the benefits of the use of such kind of tools are not yet proved to worth the cost of adoption cost.

8. Maletic, Marcus & Collard

The motivation of this taxonomy [73] was the authors' observation that existing taxonomies "are somewhat skewed with respect to current research areas on software visualization," and hence they propose a "number of realignments," in order to address tasks of real-life software engineering projects by proposing a new framework.

The framework they proposed is based on five dimensions of software visualization, each of them relating to the what, where, how, who and why of the makeup of a visualization, naming their approach a task-oriented one since it emphasizes the tasks of understanding and analysis during development and maintenance.

Their framework is based upon Card's model for visualization [74] mapping a Software Visualization process directly to it.

The five dimensions they define are:

- **Tasks**: This dimension defines why the visualization is needed, in terms of software engineering tasks, classifying also Software Visualization tools regarding the particular engineering task they support. According to the authors, "this is the driving force behind a classification of software visualization systems."
- **Audience**: This dimension defines who will use the visualization in terms of users' skill, experience, etc.
- **Target**: This dimension defines the aspect of the software that will be represented. The authors consider as targets of visualization "the architecture, the design, the algorithm, the source code, the data, execution/trace information, measurements and metrics, documentation, and process information," but also "measurements and metrics obtained from software, process information and documentation," or even "attributes relating to issues such as data collection (i.e., time of collection, method of collection, invasiveness etc.) and issues relating to the programming language and environments (e.g., paradigm, concurrency, parallel processing, etc.)"
- **Representation***:* This dimension defines how the visualization is constructed based on the available information and is closely

related to the visual metaphor that is going to be used, the navigational aids for the user, etc.

- **Medium**: This dimension concerns the medium that is going to be used to represent the visualization, knowing that different media (e.g. paper, single monitor, multiple monitors, etc.) offer different abilities for visualization.

C. COMMENTS ON EXISTING TAXONOMIES

Looking back in time, we can see that the term programming was used extensively to describe the process of creating computer applications. As the software's complexity increased, programming has become a part of Software Engineering reflecting only one step in the whole process of software construction. This change in concepts is well reflected in the existing taxonomies. The oldest taxonomies started with a narrow scope of objects to be visualized and each new taxonomy proposed was a shift towards modern methods of Software Engineering. For example, Myers taxonomy only addressed program visualization and later algorithm animation was added, while the later taxonomy of Maletic et al. is referring to all the software artifacts and the tasks during the Software Engineering process.

Another thing that we mention is that the complexity of the criteria used to classify Software Visualization systems, were changing over time with Myers proposing four categories and later taxonomies significantly raising the number of concerns in order to characterize a Software Visualization Tool. This evolution was necessary as new software visualization systems were emerging with a wider variety of capabilities and features, hence leading to more complex and detailed taxonomies.

Another issue is that the majority of the existing taxonomies were made to classify Software Visualizations tools and not to be used as taxonomies for the

area itself. As such, they mainly represent the state of the art of the Software Visualization systems of their era, even if sometimes the approaches followed are tightly coupled to specific problems the authors faced during their research and development of Software Visualization tools and the solutions they proposed.

In general, all of the existing taxonomies have their own merits and undisputable value, and have served well the area being the inspiration for a number of tools On the other hand, they do not reflect and do not cover the state-of-the-art in software development with the exception of the taxonomy proposed by Maletic et al., which is based on a solid reference model and can clearly be categorized as an effort to describe the area and not specific tools, even if this is done in a very generic, “essay like” way.

More specifically, Myers’ taxonomy is exactly what is proposed; “Taxonomy for Program Visualization Systems” with the terms program and Visualization defined in the context of this era, namely, program was a term referring to what today is called source code and Visualization was a term derived from the use of graphics to represent programs. Moreover, distinguishing them among four categories (whether the displays offered by the systems are static or dynamic and whether they visualize code, data or algorithm) is a correct but rather rough classification, compared to today’s tools and practices.

Regarding Roman and Cox’s taxonomy, based on the given definition of Program Visualization and the approach to the term visualization at the beginning of their paper, they also restrict both the input and the output of the Software Visualization process, assuming that the source of Program Visualization is a program in textual form and the result of Program Visualization is always pictures.

Nevertheless this taxonomy is the first one that is actually a taxonomy of the area and not of the various tools; it’s a taxonomy based on classification principles and not on tool characteristics and as such it is consistent with the

authors' purpose while simultaneously making clear that a tools classification can be easily derived based on the domain, the range and the nature of mapping they support. It also addresses concepts that were not present in previous taxonomies, such as the different roles of the humans that participate in the process of visualization, the various level of abstractions at which a program can be seen and the well defined abstract taxonomical criteria that satisfactorily span the area. We believe that this is the first real taxonomy, applicable directly to the area and based on long-lasting principles and not characteristics or attributes of the Program Visualization tools.

Stasko and Patterson's taxonomy has many similarities with the previous one, and, taking into consideration that it was published the same year as the previous one and also that none of the authors refer to the other work, we assume that they have come to their results separately. Even if this taxonomy is more descriptive than the previous, we again see concepts like scope and abstraction having a central role and defining axes of classification by themselves. We should also mention that the authors point out that the software visualization tasks should be structured and further analyzed. Another important thing is that the notion of "views" is also widely used throughout the paper, denoting them as goals of every visualization tool, analyzing them further and revealing the various techniques that can be used. On the other hand, there are terms that are used in a slightly confusing way such as the aspect, which is not used purely to define what will be visualized but also includes why the visualization is created ([70] page 4).

The first taxonomy proposed by Price et al. revealed the fact that we need multi-dimensional concepts in order to categorize Software Visualization systems using a set of thirty characteristics for evaluating them. It also introduced the need for a more principled and abstract way for approaching this problem, even though some of the proposed criteria are based on features and not principles.

A significant fact about this taxonomy is that, like the previous one, is not a taxonomy of the area but rather "a taxonomy we propose for characterizing

program visualization systems ([67] page 3)" and of course serves the author's motivation to "span the space of important distinctions between systems and allow us to discover where previous systems has succeeded or failed ([67] page 2)." It could easily be argued that by separating the tools among each other, we also divide the area itself. We will agree that this may be partially correct but does not apply in general and in principle.

On the other hand, categorization of tools based on what platform they run on is irrelevant to a principled taxonomy of the area. Desirable characteristics of a tool in terms of usability, performance, etc. and the degree they are present in a specific tool, should not be part of a taxonomy of the area. In addition we mention that the developer of a SV tool may choose to create a tool that produces different kind of visualizations for different artifacts and as such its functionality may span different sub-areas of the area, hence existing tools' functionality is only an indication of the area and not a criterion by itself. Hence, this taxonomy may be viewed only as a SV tools taxonomy and not a SV taxonomy, since the categories resulted "are called characteristics because each of them characterizes a program visualization system in a particular way ([67] page 3)."

A secondary observation for this taxonomy is that even if the authors adapt the term software, they limit software to programs, data and algorithms; they place the time that visualization data will be gathered, in the same level with code and data visualization, under the general category "What gets visualized?"; and that the authors point out that other modalities can be used to stimulate all the senses of the user even though they classify all those modalities in the same level with the ability of a tool to visualize using color, decreasing in this way the significance of these modalities.

Their updated taxonomy is arguing directly that the term visualization should not be considered as restricting to visible images and they refer to the seventh definition found in [46] to amplify their position. It also contains a definition that has served the area for many years and has been massively

referenced, but also reflects the technological achievements of the era and restricts the possible use of SV. One weak point is that the authors claim that a Visual Language is a kind of "*weak version*" of SV but the way it is used, i.e. "to facilitate specification rather than understanding," separates the two concepts and hence, for the SV tools that they present later, a strange characterization is introduced for those tools: "*systems employing intentional SV.*" The reference model and the conceptual basis they take into consideration for the derivation of their top level categories treats Software as a black box that is observed macroscopically; a model that is not suitable to encompass the evolution of the area. In general, this is another taxonomy that cannot be considered as a taxonomy of the area but a taxonomy for tools.

The Oudshoorn et al. taxonomy is one of the more interesting taxonomies, even if less referenced compared to the others, that introduces many significant concepts. The first thing is that the authors clearly state that all the previous taxonomies deal with the categorization of tools and not of the area itself and consequently they propose the first taxonomy that deals with the area.

Their approach of viewing the area through three different aspects is really interesting, especially if we consider their grouping of the various categories of past taxonomies, into purposes, mechanisms and ideals. For example all the required properties of a tool, in order to be useful, are nicely grouped under the category "ideals" or the various techniques that could be used under the term "mechanisms." We believe that this way of spanning the area constitutes the basis of a taxonomy for the area that can be characterized as mutually exclusive. Even if their sub-categories cannot be considered as complete, after a post-mortem analysis taking into consideration the effects of technological advancement and the evolution of the area, they still constitute a new approach for the area, at the time it was published, that can be easily expanded to accommodate new additions.

The major disadvantage of this proposal, is that it can't be considered as complete, since there are areas not included, such as the various aspects of programs that could be visualized, i.e. it does not answer the question "what is to be visualized?"

Tilley & Huang's taxonomy, is basically a taxonomy for PV techniques that also points out a use of SV in a way not mentioned by previous taxonomies but reelecting reality; as a means for documenting software systems. A very significant conclusion from this fact is that the definition of SV should be very generic regarding its potential use, since we never know where it can be used in the future.

Finally, the taxonomy proposed by Maletic et al., is the only approach dealing purely with the area and is not "tool oriented." The authors had tried to answer the same basic questions as previous taxonomies, but in a more abstract and descriptive way. Still, there are questions that are not addressed such as "How will the visualization be constructed?" (in terms of the underlying mechanics and not in terms on how to present it to the user, with the latter being addressed), "How the required data will be extracted from the underlying software artifacts?", "What are the user goals that the final visualization should support?" Their division of the area of Software Visualization, having as their basis the real task that needs to be accomplished, is a very effective, but rather abstract one. Even if they point out the significance of the roles of the users and the tasks they have to accomplish, the importance of the medium in which the representation will be rendered and many other crucial issues, they do not offer the required sub-categorization, making this taxonomy a rather descriptive and generic one, not capable of being used as a solid framework but only for triggering of new views for the area.

We believe that the existing taxonomies do not efficiently cover the area of Software Visualization in the context of software development based on modern Software Engineering principles and practices and do not reflect the reality of software construction at large. They do not address problems faced during the

software development process, such as software evolution, software metrics etc., and most of all, they do not give the proper attention to the issues related to the user, like the various mental models that describe people involved in software development or the intents and the expectations of the users.

We believe that existing taxonomies, are either very abstract and not complete or they were biased from their effort to categorize the existing tools, not actually describing the area itself, and there isn’t a single taxonomy, that contains all research questions of our field, and also connected with the everyday practice of Software Development under the prism of modern Software Engineering practices. The area of Software Visualization is still lacking a common conceptual framework that will encompass all concerns, methods and techniques and that will be used as the basis for further exploration but also as a deposit place for all existing techniques and ideas. This lack of a solid framework leads to the inability to identify abstract similarities with problems already faced within relevant research areas so that existing solutions or ideas could be transformed properly and applied to our discipline.

IV. SOFTWARE VISUALIZATION TOOLS

A. INTRODUCTION

During our research on existing SV tools, we realized that there exist so many of them that it would be impossible to present them all. So we decided to present a subset only, based on their historical significance or the new concepts and approaches they represent. Another important issue is that some of the systems presented herein, are based directly on the taxonomies proposed by their authors. Our description is brief, but we provide all the necessary links for each tool, so that the reader who would like to have a better insight can find them easily.

The most important thing is that all visualization systems are unique entities, with their own merits, and at the current state, our intention is only to present them, so that we gain an insight on what aspects of software have already been targeted by the existing systems and the various approaches that have been explored so far.

B. VISUALIZATION TOOLS

1. Sorting Out Sorting (1981)

Sorting out Sorting, was created by Ronald Baecker in 1981 [75], and is actually a video which uses animation of program data combined with an explanatory narrative that illustrates and compares nine sorting algorithms as they run on different data sets. This effort was among the first attempted to dynamically visualize the execution of algorithms and is considered the first major example of Algorithm Animation, having also practically proven the pedagogical value of animation in Computer Science.

For a more thorough review of this effort the reader is referred to [21] & [75] while a screenshot of the movie is shown in Figure 13.

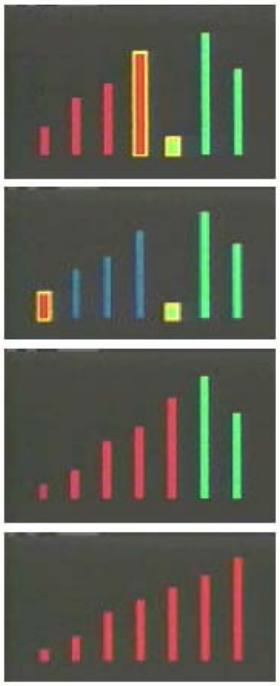


Figure 13 Linear Insertion: a) first comparison of the 4th pass, with the first 4 items already correctly ordered; b) final comparison of the 4th pass; c) end of the 4th pass, after the 5th item has been moved to the front; d) data is sorted. Colors (shown as gray scale) denote "unsorted" and "sorted," i.e., in the correct position thus far. Borders indicate that two items are being compared [75].

2. BALSA (1983) & BALSA II (1988)

Marc Brown and Robert Sedgewick developed an interactive software visualization system named Brown University Algorithm Simulator and Animator (BALSA), in 1983 [25] & [76] that was successfully used as a teaching aid in Computer Science classes at the same university for a number of years.

This system was able to display, in black and white, multiple simultaneous views of the animated algorithm's data structures, and also multiple algorithms executing at the same time. Additionally the user was able to control the speed and the direction of the animation of the algorithm and also a pretty-printed

display of the source code was displayed, indicating also the currently running line of code. An example of a visualization produced by BALSA is shown in Figure 14.

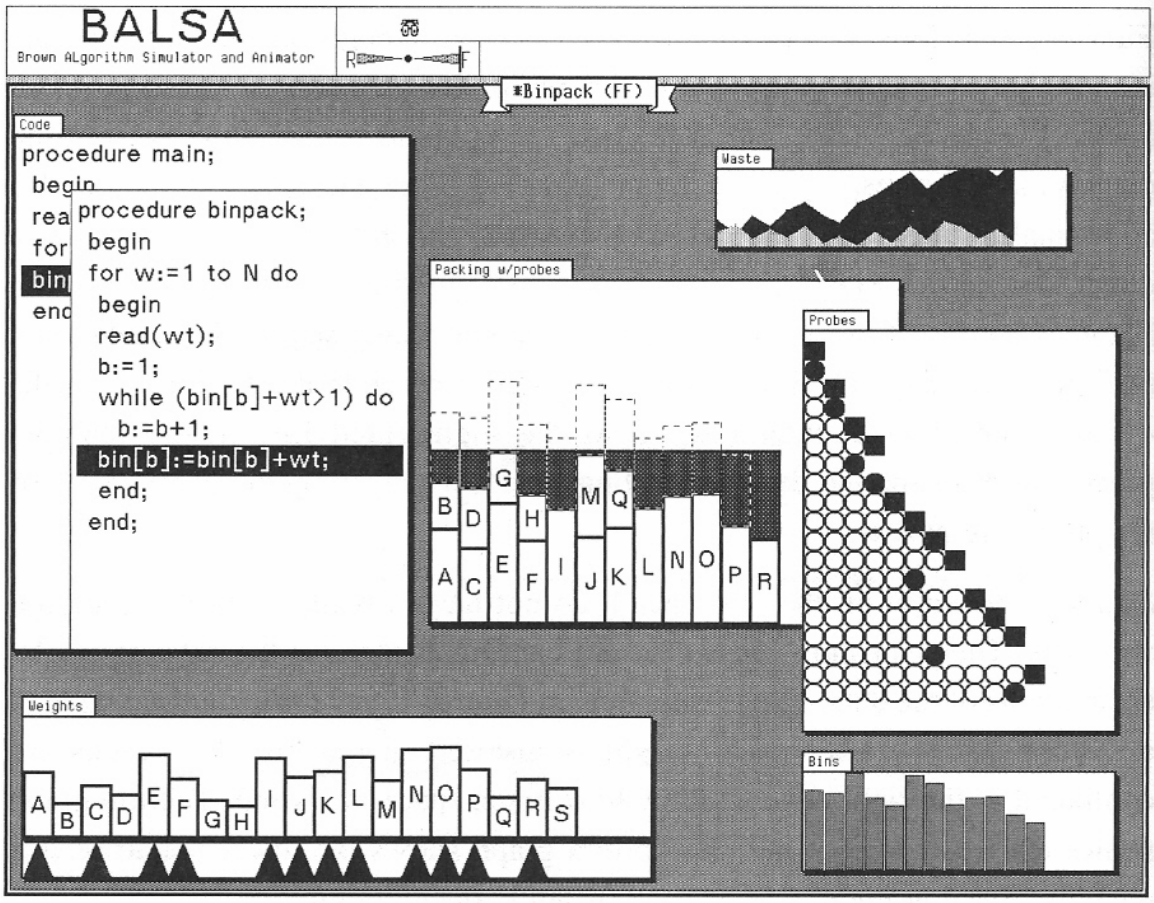


Figure 14 First-fit binpacking algorithm as visualized by BALSA [80]

BALSA introduced the use of multiple views but, more important, the model of “interesting events.” The concept behind “interesting events” is that there are specific points (like the entry into a function or the access of a particular data structure) that are of great interest during the execution of the algorithm. This way, the typical procedural way to visualize the algorithm was substituted by a higher level of visualization, where specific parts of the code were encapsulated into meaningful operations. It is obvious that the visualization of the

whole procedural process could be achieved by assuming that each line of code is an interesting event. A detailed explanation of interesting events is provided at [77].

In 1988, Brown released Balsa-II [78] & [79], which provided additional scripting facilities giving the user the ability to annotate the implementation of the algorithm providing his own interesting events to be visualized, color animations and the ability to use sound in addition to visual representations.

An example of visualization produced by BALSA-II can be seen in Figure 15

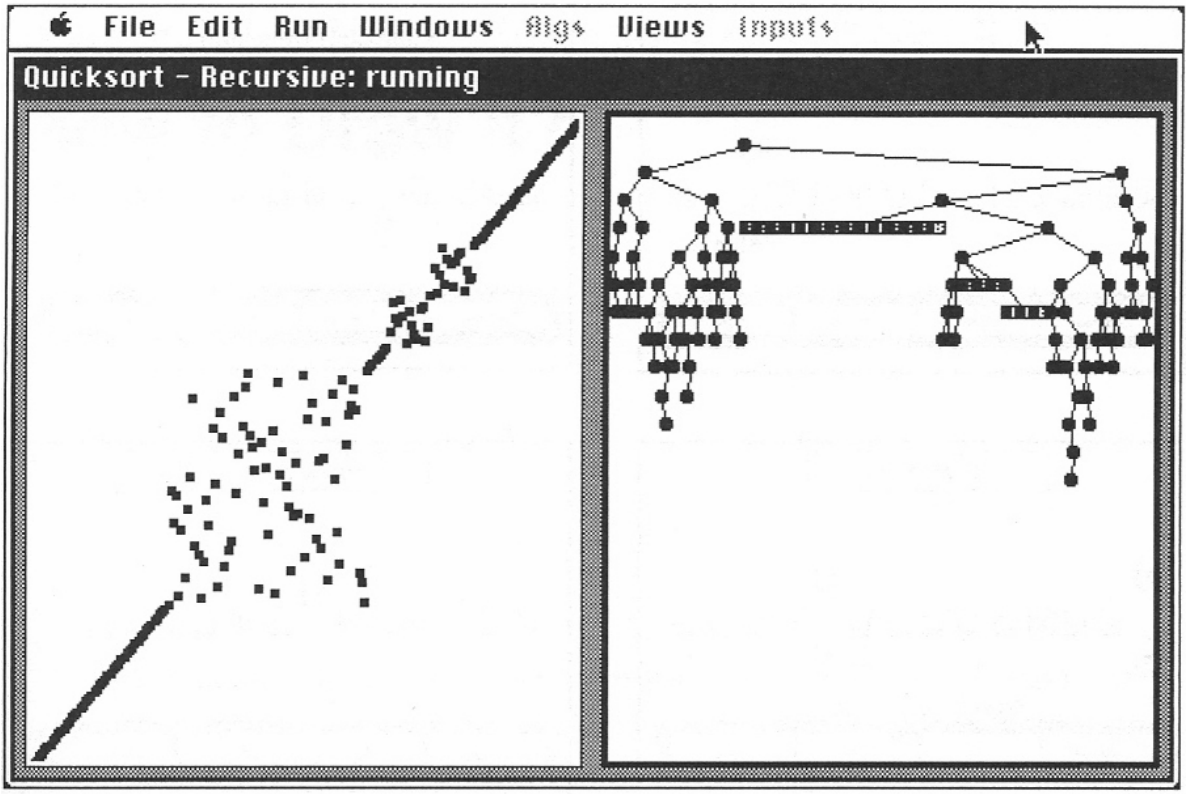


Figure 15 Quicksort in action in BALSA-II [80]

3. Tango (1990) – XTANGO (1992)

TANGO (Transition-based Animation GeneratiOn), was developed by John Stasko at the Brown University, introducing the path-transition paradigm for animation design and a framework for algorithm animation systems [81] &

[82], adopted by many later systems as their fundamental architecture, since it allowed for smoother animations and less overhead for the visualization designer.

The path transition paradigm's supporting architecture is based on the idea of manually instrumenting the source code (in this case C) with special calls that define transitions rather than steps, providing smoother animation. These transitions are defined in terms of trajectory, size, visibility, and color.

Another significant characteristic of TANGO was the fact that it allowed a many-to-many relation between program source and the animation system, since it had the ability to receive events from several different program sources while a program source could supply events to several different animations. From a usability point of view, TANGO offered basic control of the view, allowing the user to pan, zoom and pause the animation.

X-TANGO was a descendant of TANGO, being a version of the latter created on top of UNIX and the X11 Window System, that also included a large number of predefined algorithm animations [83]. A snapshot of an animation produced by XTANGO is shown in Figure 16.

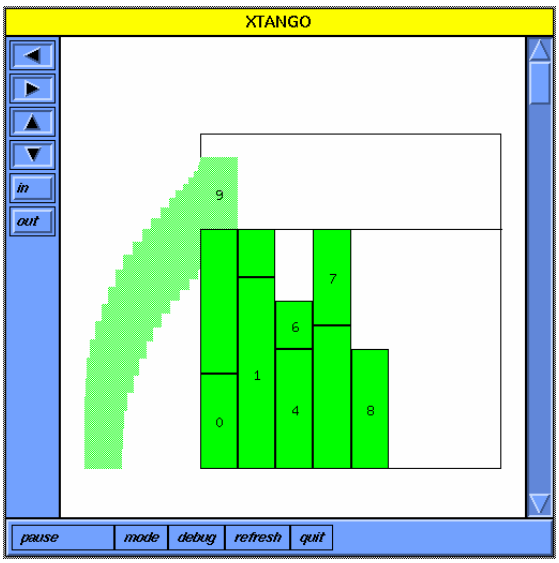


Figure 16 Snapshot if an animation for binpack produced by Tango [0]

For the interested reader, more info on XTango, and a downloadable version, can be found at the site of Georgia Institute of Technology [0].

4. Polka (1993) – Samba(1996)

POLKA was another tool introduced by John Stasko with its original version implemented in C++ on top of UNIX and the X11 Window System and later expanded in POLKA-3D [86] & [87].

POLKA offered an improved animation design model (requiring location, animation object and action to be performed), introduced explicit animation time, multiple animation windows and rich visualization / animation capabilities and semantic zooming, as shown in Figure 17.

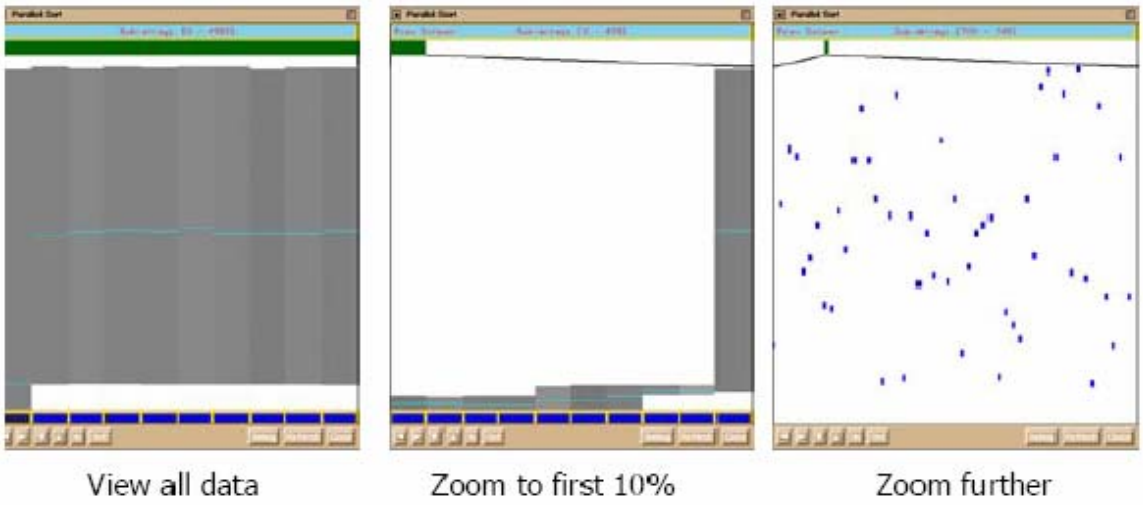


Figure 17 Semantic zooming in POLKA family [88]

It can be used both as a general-purpose animation system for algorithm and program animations but also for animating serial programs. POLKA is a descendant of XTango system providing a more user-friendly interface and its own high-level abstractions.

Samba was introduced some years later as an interactive, front-end graphical interface [89] but also added the important ability to read a series of simple ASCII commands with parameters (e.g. rectangle 3 0.1 0.9 0.1 0.1 blue

solid) that will be used to direct the animation, making Samba an interactive animation interpreter, able to visualize programs in any programming language.

Later on, a Windows-native version of Polka, called PolkaW, was developed along with a Java version of Samba, called JSamba. More information on those tools are provided at Graphics, Visualization, and Usability Center, Georgia's Institute of Technology site [0] while some examples of visualizations offered by those programs are shown in Figure 18.

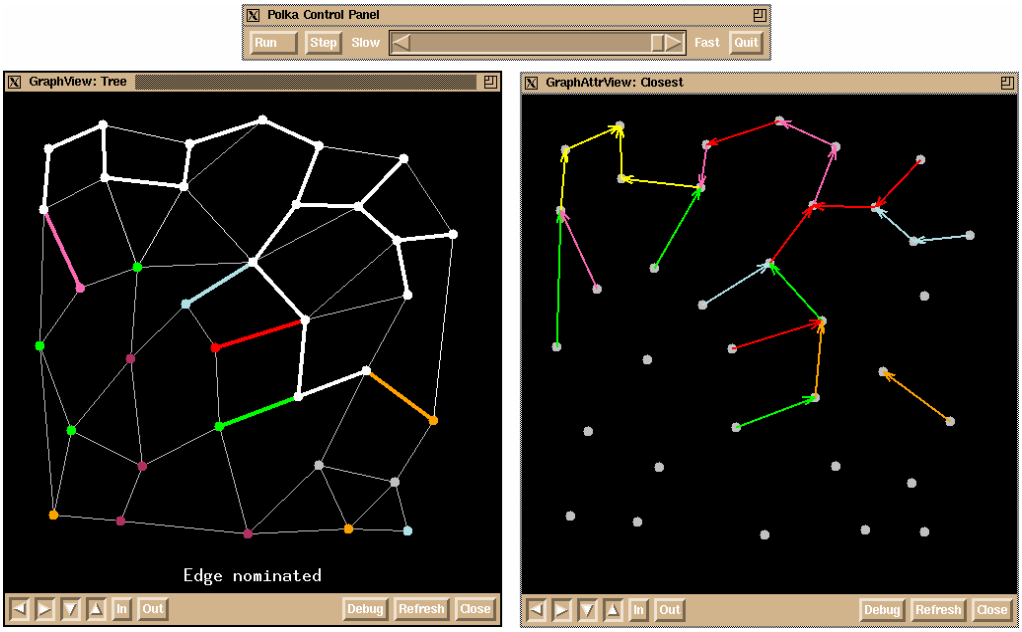


Figure 18 POLKA's animation of a parallel minimum spanning tree program. The left view shows the graph and the spanning tree growing inside it. The right view shows the "closest" data structure maintained by the program [91].

Polka was later extended to provide 3D views using the Iris GL 3D Graphics library offering 3D primitives to the creator of the animation and the ability to control the position and orientation of the viewpoint.

5. Zeus (1991)

Another descendant of BALSA is Zeus, also developed by Brown [92], [93] & [94] that provides more control of the data during runtime, support of program auralization using non-speech sound using the MIDI interface, a control panel as

an interface to the user offering many configuration facilities (start, stop, stepping, speed control, etc.) over the algorithms to be displayed and even the ability to create snapshots of the system's current state that can also be used as restoration points for future executions.

Zeus targeted mainly computational geometry algorithms, operating systems algorithms, hardware design algorithms and also multi-processor, multi-threaded platforms. The data for the visualization produced were based on the annotation of code with interesting events. Examples of animations created with Zeus are shown in Figure 19.

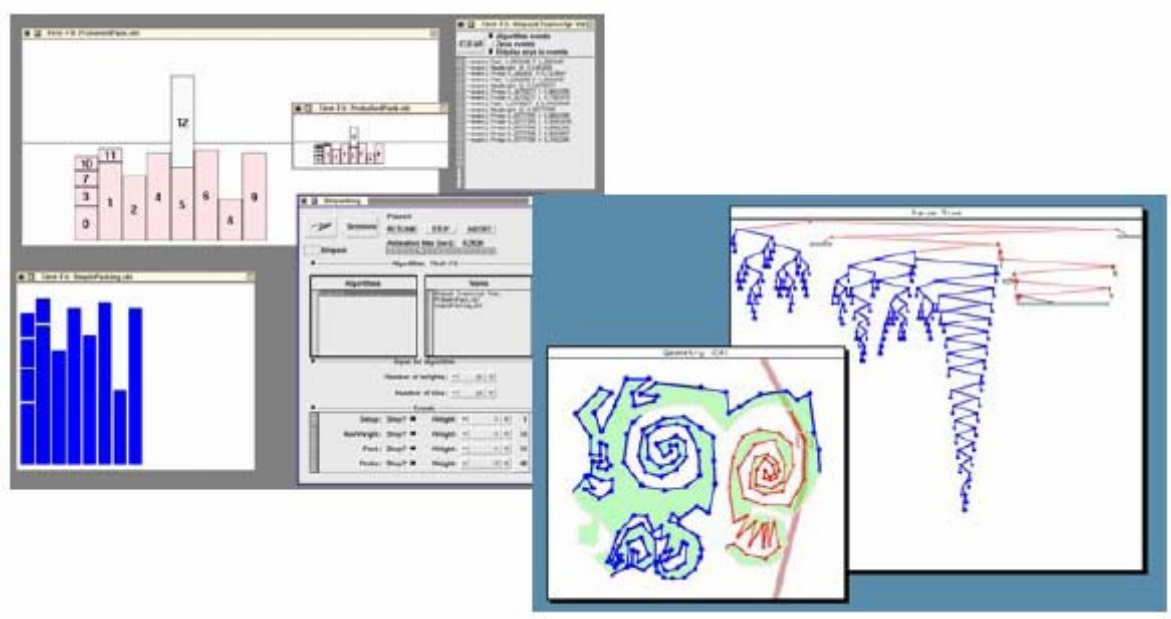


Figure 19 Algorithm animations produced by Zeus [88]

In 1993, Brown and Nojark, extended Zeus to include 3D viewing capabilities [95]. The 3D version of Zeus includes three dimensional primitives, 3D navigation through mouse interaction, as well as control panels to change rendering parameters and view specific parameters.

6. SeeSoft (1992) – SeeSys (1994)

SeeSoft, created by Stephen Eick, Joseph Stefen & Eric Summer Jr in conjunction with the 5ESS @Telecommunications Switch Project [96], was the

first tool to implement the representation of each line of code as a single colored line or a single colored pixel on the screen. Each line on the screen could be of a fixed size or with a length depending on the number of the characters contained in the respective line of the source code. A typical display created with Seesoft can be seen in Figure 20.

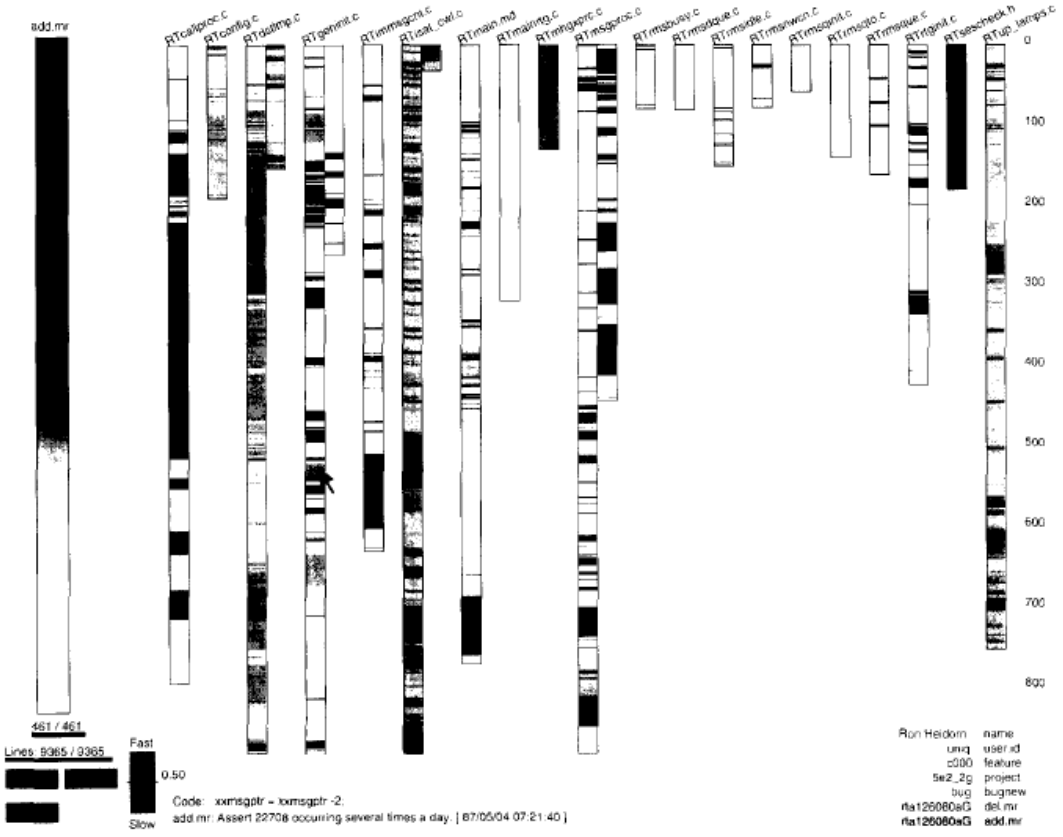


Figure 20 A display produced from Seesoft displaying non indented code for different files, showing the relative size of the files, the age of code and how many times a file has been changed [96]

This representation offers a significant reduction of the item to be visualized, increasing the capacity of the screen offering at the same time increased readability through the coloring properties and the advanced facilities

for direct manipulation, offered by the program. On the other hand the system had a limitation in scalability since it was able to display up to 50,000 lines of code on a 1280x1024 monitor.

Another interesting characteristic was that it offered two levels of abstraction; the line representation and the display of the source code itself in a separate “code reading” window opened by the user as a result of a magnification of a specific area of the line representation. The approach proposed by Seesoft was very expressive and is still influencing Software Visualization tools today; it was similar to the view someone may get by printing all the source code, placing one file next to the other on a wall, and then walking away looking at the printed code!

SeeSys [97] is the successor to SeeSoft, based on the same reduced representations, providing a space filling technique for displaying source code related statistics, generalizing the same techniques based on Johnson and Schneiderman’s work [98] on visualization of hierarchical data using treemaps. Motivated by a large communications software system, with several million lines of code are organized hierarchically into tens of sub-systems, several thousand directories, and hundreds of thousands of files, developed and maintained by AT&T, it is one of the few tools that found practical use in every day Software Engineering.

The major benefit of SeeSys was that it related the statistics for the software to specific components, in this way placing the statistic in context. As shown in Figure 21 the key idea was to represent the system’s structure using rectangles that would be later filled with useful statistical information, offering a straightforward visual comparison.

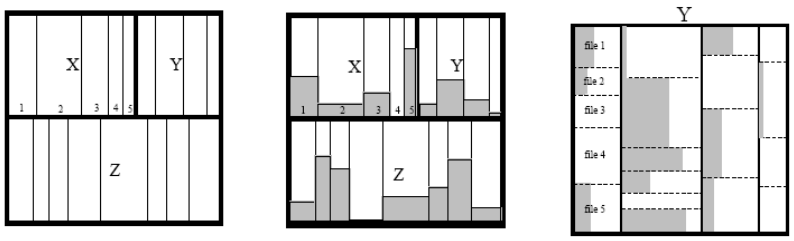


Figure 21 Left pane: subsystem and directory statistics. Middle pane: a fill statistic for directories. Right pane: a zoomed view on subsystem Y showing file level statistics [97].

A typical screen produced by SeeSys can be seen in Figure 22. The system also offered a significant number of facilities for user interaction and techniques, making the tool very efficient.

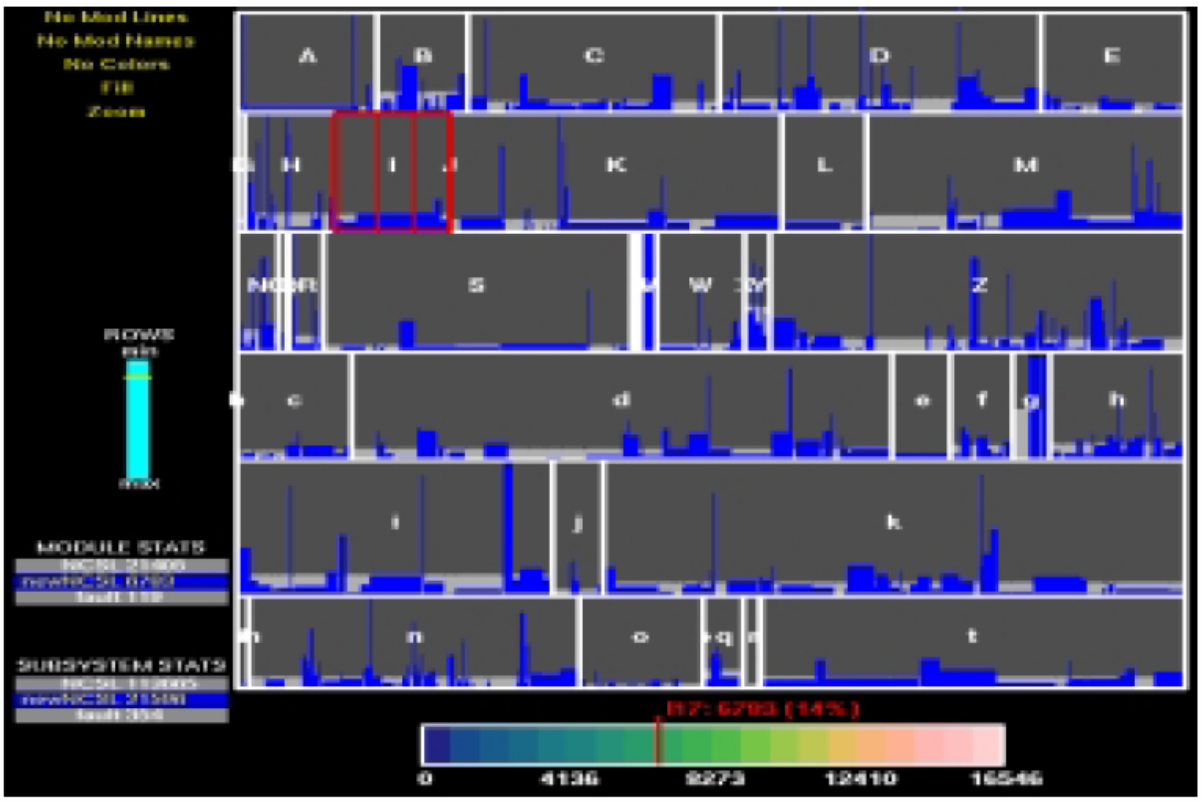


Figure 22 Bug rates by sub-system and directories as presented by SeeSys [97]

A later version of SeeSoft was intended for Web-based analysis of Large Scale Software systems [99]. In this version, SeeSoft is a part of a suite of tools, that aim to assist describing and understanding different aspects of software evolution by examining changes to documents and visualizing system artifacts such as source code and source version history. SeeSoft is implemented as an applet-based source code visualization system, based on the same reduced representations. A screen produced by the web-based version of SeeSoft can be seen in Figure 23.

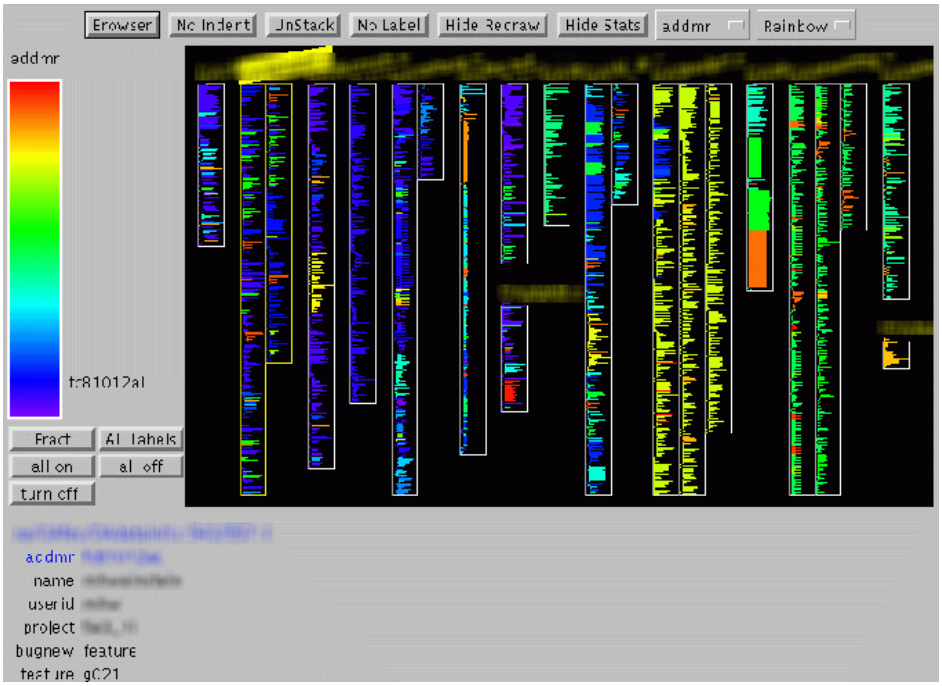


Figure 23 The SeeSoft text view showing code age according to a rainbow color scale [100].[11]

7. CVSscan

This is a tool used to visualize the evolution of source-code structure and attributes based on a line-based approach, similar to the one followed by SeeSoft, targeted for a variety of roles during the software.[12] It is based in a

[11] Proprietary information has been blurred in the figures

[12] CVSscan is a part of the Visual Code Navigator suite, which contains four stand-alone applications. For more information, the interested reader is referred to the official web site of this suite @ http://www.win.tue.nl/~lvoinea/VCN.html. last date accessed?

concrete data model that classifies the status of line codes throughout a project life cycle as constant, modified, deleted, inserted, modified by deletion, or modified by insertion, and then using a 2D layout on a single display for a file's entire evolution based on a fixed-length pixel line for all code lines, as seen in Figure 24.

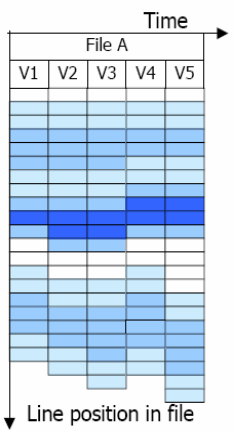


Figure 24 CVSscan representation approach for different versions of the same file over time [101].

To represent the various source code attributes, they use a variation of colors as shown in Figure 25, with one color scheme active each time.

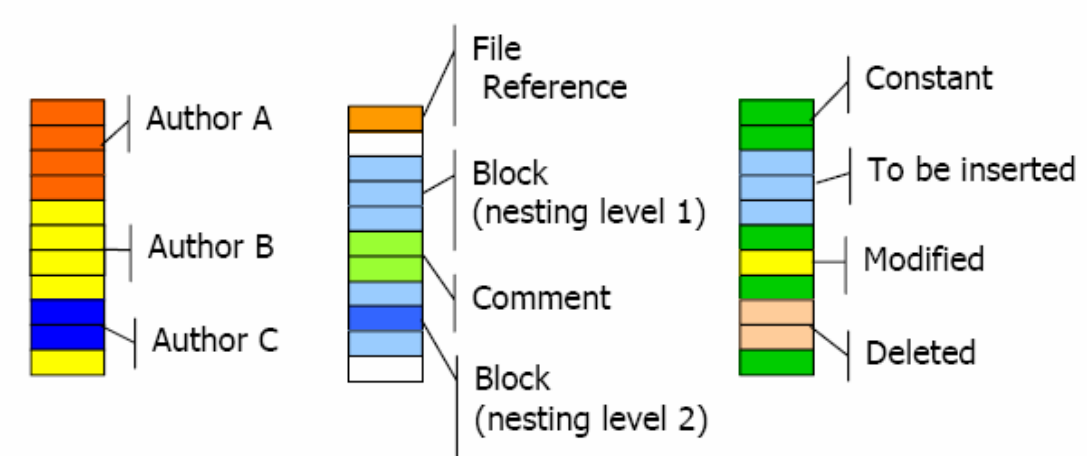


Figure 25 Colors used for encoding source code attributes. Represented (from left to right) are author, construct, and line status [101].

For the vertical layout they use two approaches, the first is a file based with each line in the y axis representing a specific line number in each file and the second is a line based layout where each line in the y axis is the global line's position (a unique label for every line written) offering two distinct views of the source code as seen in Figure 26.

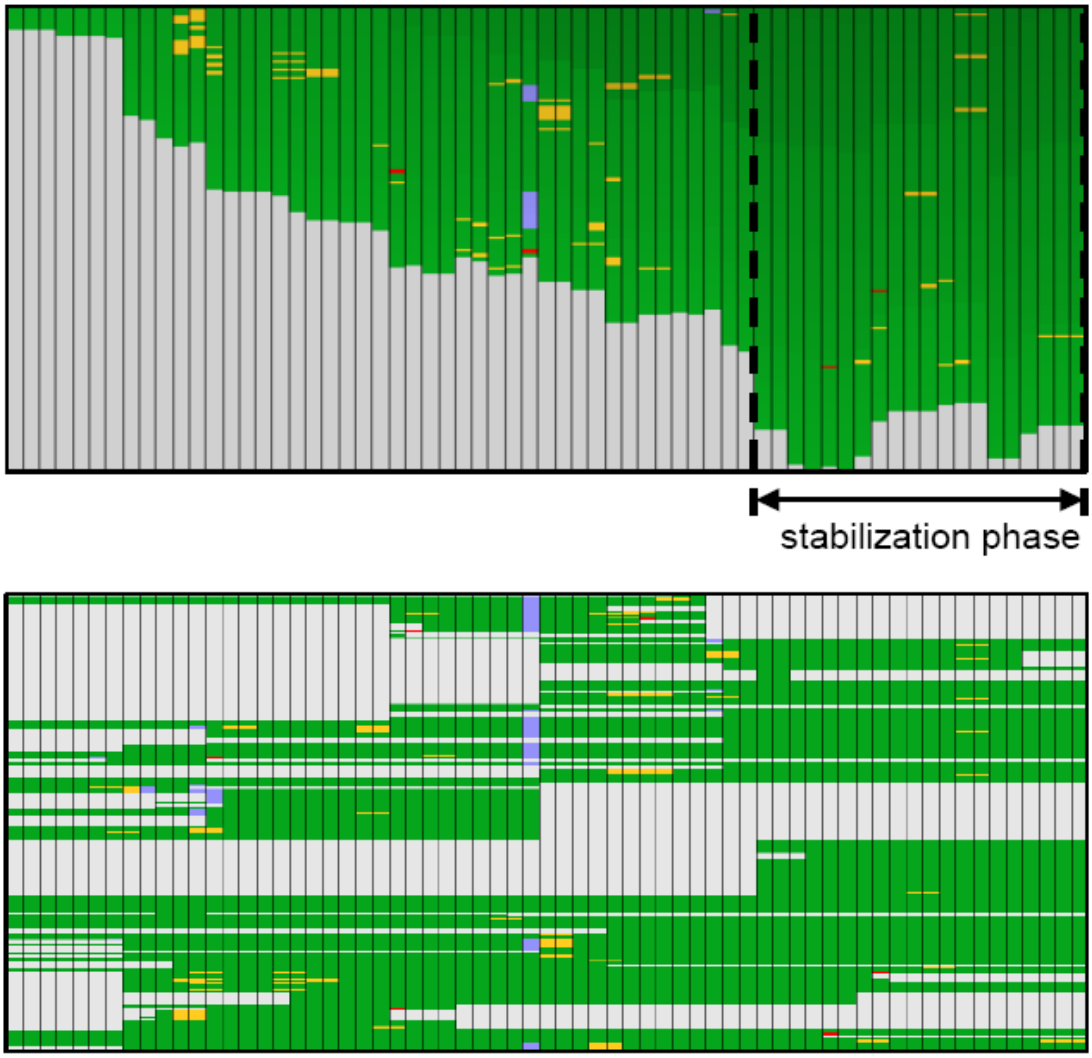


Figure 26 File based (top) and line-based (bottom) layouts of CVSscan for a file with sixty five versions [101].

One of the most powerful characteristics is the multiple views that CVSscan offers. More specifically, it offers two additional metric views and a text view on selected code fragments as seen in Figure 27, offering also significant facilities for user interaction.

Figure 27 Multiple code views in CVSscan[101].

8. SEE

This attempt represents a different approach for program visualization. It is based on the observation that the textual program appearance has changed little since the first high-level languages were developed, with the simple ASCII representation being the dominant one. Their effort was focused on the visual appearance of traditional textual program representations, using a UNIX-based system for typesetting programs written in C [19], by combining human factors research with typography principles. The result of this typesetting process was a

kind of “program book” with cross-references and indices that facilitate navigation through the source code, as can be seen in Figure 28.

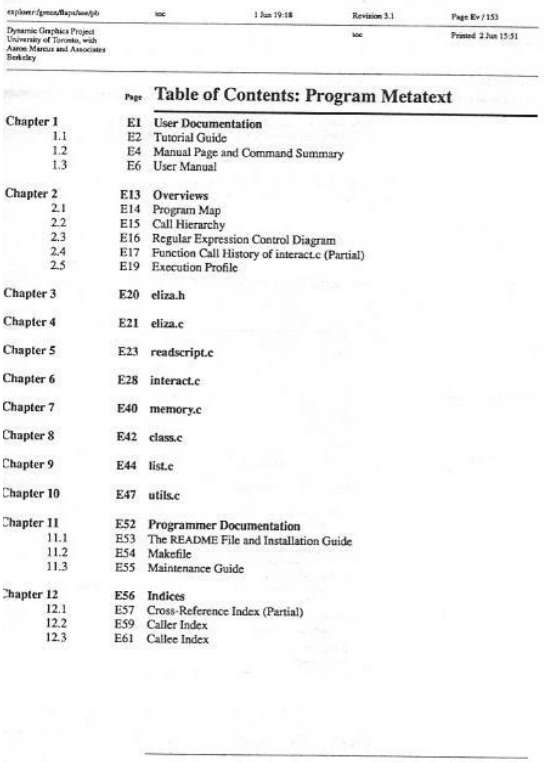


Table of Contents: Program Metatext

	Page	
Chapter 1	**E1**	**User Documentation**
1.1	E2	Tutorial Guide
1.2	E4	Manual Page and Command Summary
1.3	E6	User Manual
Chapter 2	**E13**	**Overviews**
2.1	E14	Program Map
2.2	E15	Call Hierarchy
2.3	E16	Regular Expression Control Diagram
2.4	E17	Function Call History of interact.c (Partial)
2.5	E19	Execution Profile
Chapter 3	**E20**	**eliza.h**
Chapter 4	**E21**	**eliza.c**
Chapter 5	**E23**	**readscript.c**
Chapter 6	**E28**	**interact.c**
Chapter 7	**E40**	**memory.c**
Chapter 8	**E42**	**class.c**
Chapter 9	**E44**	**list.c**
Chapter 10	**E47**	**utils.c**
Chapter 11	**E52**	**Programmer Documentation**
11.1	E53	The README File and Installation Guide
11.2	E54	Makefile
11.3	E55	Maintenance Guide
Chapter 12	**E56**	**Indices**
12.1	E57	Cross-Reference Index (Partial)
12.2	E59	Caller Index
12.3	E61	Callee Index

(a)

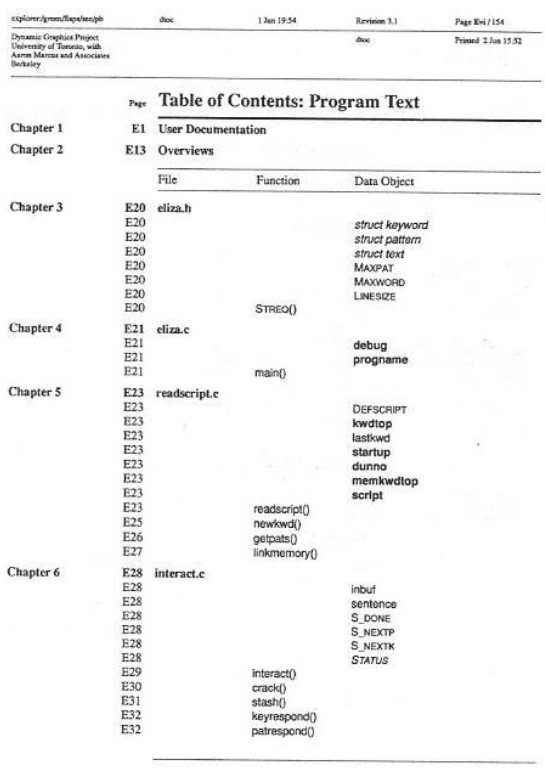
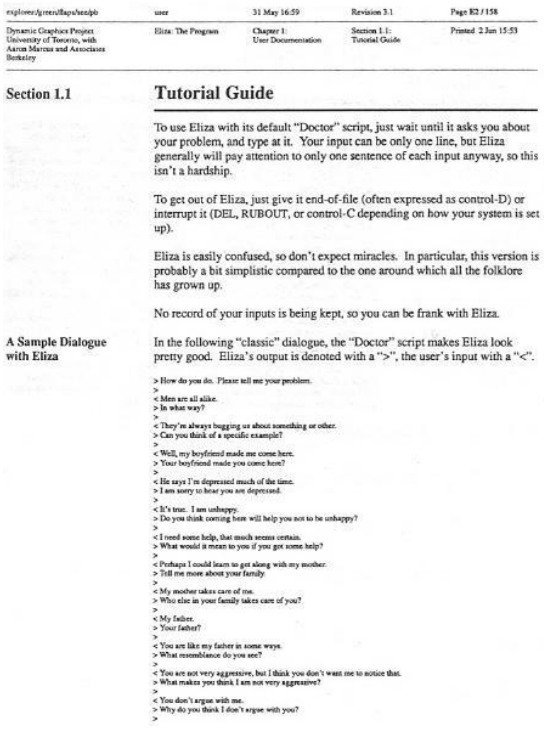


Table of Contents: Program Text

	Page			
Chapter 1	**E1**	**User Documentation**		
Chapter 2	**E13**	**Overviews**		
		File	Function	Data Object
Chapter 3	**E20**	**eliza.h**		
	E20			*struct keyword*
	E20			*struct pattern*
	E20			*struct text*
	E20			MAXPAT
	E20			MAXWORD
	E20			LINESIZE
	E20		STREQ()	
Chapter 4	**E21**	**eliza.c**		
	E21			**debug**
	E21			**progname**
	E21		main()	
Chapter 5	**E23**	**readscript.c**		
	E23			DEFSCRIPT
	E23			**kwdtop**
	E23			lastkwd
	E23			**startup**
	E23			**dunno**
	E23			**memkwdtop**
	E23			**script**
	E23		readscript()	
	E25		newkwd()	
	E26		getpats()	
	E27		linkmemory()	
Chapter 6	**E28**	**interact.c**		
	E28			inbuf
	E28			sentence
	E28			S_DONE
	E28			S_NEXTP
	E28			S_NEXTK
	E28			*STATUS*
	E29		interact()	
	E30		crack()	
	E31		stash()	
	E32		keyrespond()	
	E32		patrespond()	

(b)



Section 1.1 Tutorial Guide

To use Eliza with its default "Doctor" script, just wait until it asks you about your problem, and type at it. Your input can be only one line, but Eliza generally will pay attention to only one sentence of each input anyway, so this isn't a hardship.

To get out of Eliza, just give it end-of-file (often expressed as control-D) or interrupt it (DEL, RUBOUT, or control-C depending on how your system is set up).

Eliza is easily confused, so don't expect miracles. In particular, this version is probably a bit simplistic compared to the one around which all the folklore has grown up.

No record of your inputs is being kept, so you can be frank with Eliza.

A Sample Dialogue with Eliza

In the following "classic" dialogue, the "Doctor" script makes Eliza look pretty good. Eliza's output is denoted with a ">", the user's input with a "<".

```
> How do you do. Please tell me your problem.
>
< Men are all alike.
> In what way?
>
< They're always bugging us about something or other.
> Can you think of a specific example?
>
< Well, my boyfriend made me come here.
> Your boyfriend made you come here?
>
< He says I'm depressed much of the time.
> I am sorry to hear you are depressed.
>
< It's true.  I am unhappy.
> Do you think coming here will help you not to be unhappy?
>
< I need some help, that much seems certain.
> What would it mean to you if you got some help?
>
< Perhaps I could learn to get along with my mother.
> Tell me more about your family.
>
< My mother takes care of me.
> Who else in your family takes care of you?
>
< My father.
> Your father?
>
< You are like my father in some ways.
> What resemblance do you see?
>
< You are not very aggressive, but I think you don't want me to notice that.
> What makes you think I am not very aggressive?
>
< You don't argue with me.
> Why do you think I don't argue with you?
>
```

Section 2.1 Program Map

Figure 28 Four miniatures pages from a C program book as published by SEE [102].

9. ShriMP – Creole

SHriMP (Simple Hierarchical Multi-Perspective) started as a visualization technique, implemented in Tcl/Tk (a scripting language) and incorporated into the Rigi reverse engineering system in order to enhance its capabilities [103]. The initial concept was based on the implementation of fisheye views of nested graphs of software artifacts. SHriMP used two algorithms for manipulating large graphs: one for the creation of the nested graphs and one for the creation of the fisheye views that addressed the lack of notion for geometric distance between nodes by uniformly resizing the nodes of less interest and hence manage screen space for the focal node to grow. An example of the way nested graphs were presented from Rigi with SHriMP can be seen in Figure 29.

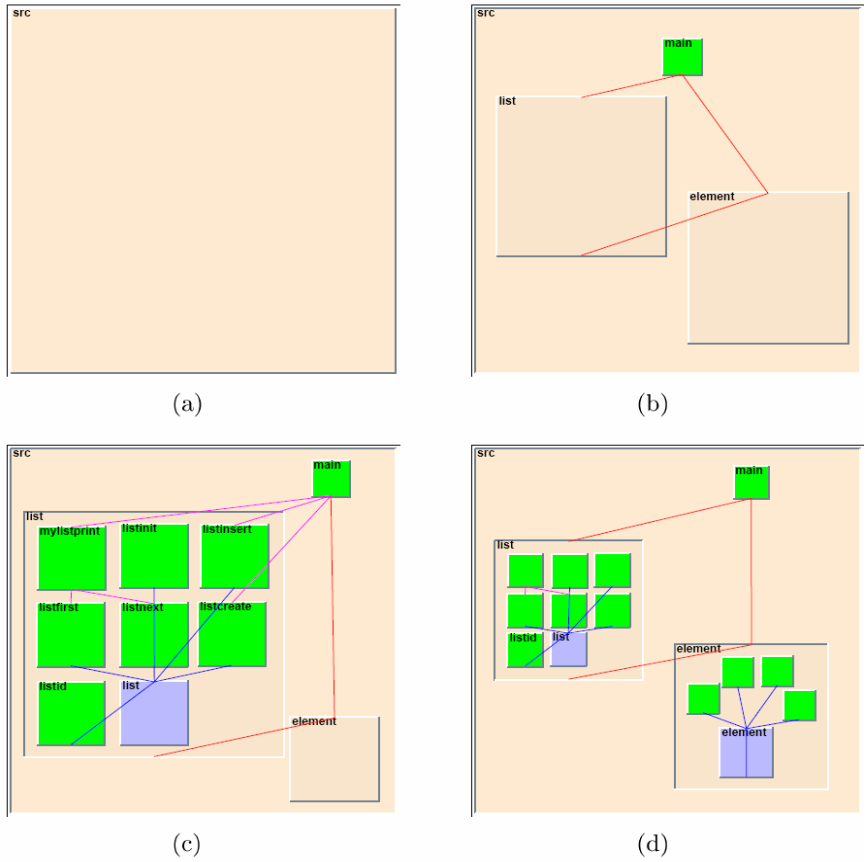


Figure 29 Opening a node, the node's contents are displayed inside the opened node and the user is descending the program's hierarchy with more details being presented while preserving the context [103].

The second version of SHriMP [104] was implemented in a graphics extention of Tcl/Tk called Pad++ and offered, in addition to the existing technique of context and detail, the ability to pan and zoom for nodes of interest. The former technique was implemented with the same fisheye view algorithm as the previous version of SHriMP that was preserving orthogonality and proximity among nodes. The latter technique was implemented in a way that zooming in a node, in the beginning context was removed from the display while at the end the source code was presented, as can be seen in Figure 30.

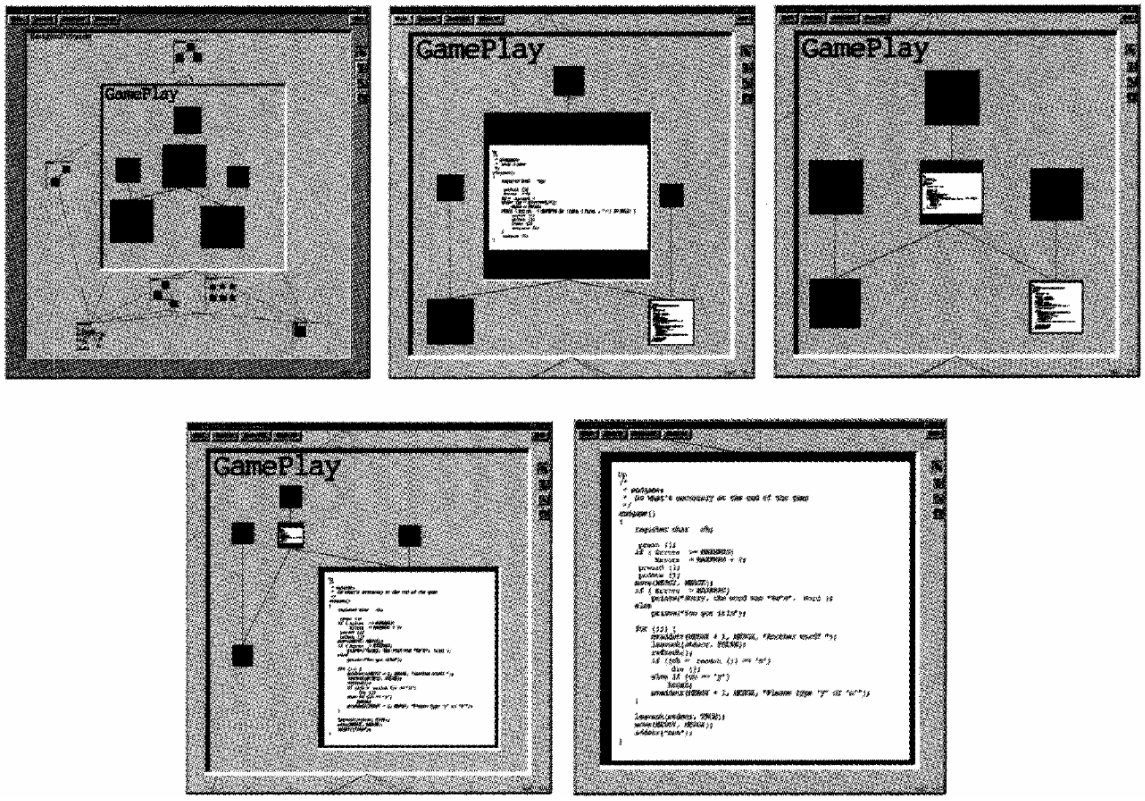


Figure 30 Magnifying a simple C program, the context is removed until the max zoom level is reached, which is the source code [104].

The evolution of SHriMP continued with the creation of a Java based stand alone application for program visualization, based on the same principles as the previous versions but with additional capabilities [105] like geometric, semantic and fisheye zooming, the integration of a search tool that offered general search, relation search, artifact search facilities and a tracing relation

component that helped the programmer to search for various relationships such as data access and method calls using horizontal tree layouts. The final two versions consisted of the implementation using Java Beans technology [106] and integration with Java Development Tools included with Eclipse [107], named Creole.

V. SOFTWARE VISUALIZATION CHALLENGES

Software Visualization, as a mean to reduce software's inherent complexity, is among the top wishes of software engineers. Looking at the answers of the survey conducted by Koschke [8] we can see that 82% of the participants stated that Software Visualization can be a significant aid for their job but at the same time, they are asking for properties not present in existing tools. A closer look at the results and the requested characteristics of Software Visualization tools, will reveal issues never addressed from existing taxonomies such as the lack of considering users' cognitive models and user needs, representations not closely related to user tasks and goals offering generic usability and not being oriented toward real problems. Participants in the survey, asked for improvements in GUI offered by the tools and ease of use, they asked for integration and interoperability, they asked for more efficient layout and use of screen space, they asked for automated matching between their intent and the visualization that is presented and much more.

These issues are among some of the significant research questions that pose crucial challenges for the future of Software. It is true that many tools have been proposed in the past, claiming that they can be used as an effective means in the fight against software's increased complexity but there is no proof that the representations they offer are superior to the traditional, textual ones. While there is no doubt that visual images can take advantage of the increased "bandwidth" of our vision, there are many times that artistically pleasing visualizations are not actually useful in understanding software.

One of the most important challenges that still remains unaddressed and, despite its significance, not included as a concern in the existing taxonomies, is the question "Which visualization is supportive for each user's task?" Matching available representations with users' tasks, goals and needs is a fundamental issue if we want our tools to be really useful. Otherwise, no matter how impressive the offered visualizations may be, they will not serve the user but will

remain technical novelties. This question is only a part of the most important factor in every tool we make or representation we provide; the user. Only the last taxonomy, proposed by Maletic et al., distinguishes the importance of the user, but still not to the proper degree.

Issues related to the tasks users have to accomplish, their goals when they are working on each of these tasks during the various phases of the software life cycle, the special characteristics of each individual user related to his expertise and his role in the development or maintenance process that induct different levels of abstractions and levels of detail, should be among the top priority research issues for our field.

Closing the linguistic gap between the user and the system, supporting users at the conceptual level in the analysis of their information needs, in the formulation of an appropriate search strategy, and in the evaluation of the obtained results area also pose challenges for the field.

As stated by Lethbridge and Singer, “many software visualization tools are developed without adequate attention to understanding the context of use for the technique, or without considering the cognitive abilities or load that is placed on the programmers by other activities [108].” We should stop expecting software engineers to adapt to our tools and try to make our tools adaptive to them. This is something that is also reflected in the existing taxonomies that serve, by definition, as the basis for the research of the area.

Our next group of challenges deals with the visualizations we provide and their effectiveness in fulfilling user needs. There are many unresolved research issues related to which types of diagrams are most appropriate for aiding program understanding, debugging and all other users’ tasks. More specifically, it is unknown exactly which forms of graphical documentation are most suitable for each users’ needs and in which specific usage context.

The layout problem is another challenge for the area; where to place the various shapes, their connecting lines and the color that should be used without

increasing the clutter and hence making a visualization useless. Aesthetics of the final product is a challenge; existing tools provide interesting and information-full visualizations, but many of them will not attracts user's attention simply because they do not pay attention to the screen's real estate problem.

Another challenge, related to the representations offered, is to find the golden mean between graphical and textual representations. We should not condemn traditional textual representations as inefficient; they have served and are serving the area of Software Engineering having proven their value. The question should be "How can we complement the traditional textual views?" and not "How do we replace the traditional textual views?" Text is a familiar representation for all human beings and we should use its power in our visualization tools without trying to exterminate it having tools offering pictures in situations where text would be more economical exaggerating the "intuitive nature" of graphics.

The interaction offered is another major challenge. Techniques like overview, zoom, filter, details on demand; requirements for customized visualizations with variable granularity offered and support of multiple views are out there but not coupled with user goals during the visualization process. Displays offered should be flexible, incorporating many different ways of visualizing the same concepts and not based on a couple different representation models. We also have to mention that many Software Visualization tools operate as standalone applications that often require data to be provided in a format different from the one developers are using.

Finding metaphors that will be able to scale and visualize large data is another challenge. Knowing also that there are often visual representations that are physically larger than the text they replace, makes the challenge more difficult to solve. Limited screen space further increases the difficulty of this problem, making it hard to present information from real programs leading to one of the most commonly cited reasons for the failure of SV systems; scalability.

Mining the required data among the existing information in various software artifacts, is another challenge of great significance. There is much more information than existing tools are exploiting, leaving this responsibility to the user.

Tools that rely on the modification of the software artifacts solely by the user are inappropriate for large software engineering projects. Of course gathering the necessary data for visualization is another challenge, along with their management and their communication throughout the various components of the Software Visualization system. Finding ways to extract the required data without interfering with software's run-time behavior is important and is a challenge on its own.

We believe that Software Visualization's research should make a shift towards a more user oriented approach paying more attention to the cognition issues of the final recipients of our products; precious "know-how" should be incorporated from relevant research areas and become the driving forces for new tools. In our era, users are demanding ease of use, better interaction, more adaptive environments, more functionality and, more important, tools that are made for their needs and not as a demonstration for another novel metaphor. A neat and clear classification of concerns is required, since existing taxonomies do not offer this.

We believe that the bigger challenge is understanding user needs and matching existing representation models to those needs. If we succeed in this, then the next step is to solve the technical issues on how to construct those visualizations combining various data sources and integrating various techniques. This is something that can be achieved only if we have a properly defined taxonomy that will provide a framework for discussion, analysis and research guidance by offering a systematic and systemic overview of the area, covering all the concerns and challenges.

VI. A MULTI-LAYERED FRAMEWORK FOR SOFTWARE VISUALIZATION

A. INTRODUCTION

Today, software is constructed in a more or less engineered fashion, in a way quite different from the past, making programming just a step of the whole procedure. There are other activities that take place such as software design, requirements analysis, computation of metrics, maintenance, etc., with none of the existing taxonomies covering issues faced during all those phases and consequently not incorporating the tools constructed to support those activities.

We propose a multi-layered approach that is able to describe the whole area of Software Visualization at an abstract or more detailed level and also is expandable, by further dividing the existing layers into sub-layers.

B. PROPOSED DEFINITIONS

1. Computer Program

Our definition will be based on the definition proposed by IEEE [31] but we will expand it so that it will be best fitted in the context of visualization and clearly separate program from software.

We will define the term “computer program” as follows:

> A computer program, or simply program, is a series of computer instructions and data definition, expressed either in a form acceptable directly by a computer (machine language[13]) or any other human-readable form that can be translated into its machine language equivalent that, when submitted as a unit, will be

[13] For the term machine language, we accept the IEEE’s definition [31] according to which a machine language is “*a language that can be recognized by the processing unit of a computer. Such a language usually consists of patterns of 1s and Os, with no symbolic naming of operations or addresses.*”

executed or interpreted by a computer hardware, enabling it to perform computational or control functions to produce a result.

2. Code - Source Code

We will fully accept and utilize the definitions provided by IEEE in [31] where code is defined as:

> (1) In software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or other translator. See also: source code; object code; machine code; microcode.
>
> (2) To express a computer program in a programming language.
>
> (3) A character or bit pattern that is assigned a particular meaning; for example, a status code.

and **source code** is defined as “Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator. Note: A source program is made up of source code.”

3. Algorithm

For the needs of this thesis, we will adopt the IEEE definition in which [44] algorithm is defined as “A finite set of well-defined rules for the solution of a problem in a finite number of steps.”

4. Software

We believe that the definition of software, as provided by IEEE [31], is sufficiently descriptive and complete and hence we will accept it and define software as “computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.”

5. Software Visualization

We believe that none of the existing definitions for Software Visualization is able to follow the evolution of the area, if we consider the recent research results. This is mainly caused by the existence of the word “visualization” that creates a biased perception for the area.

It is well known that the first steps in the area were aiming to produce visual representations of programs, their associated data and well-defined computer science related algorithms, in order to assist program or algorithm understanding. Later on, tools that uses also other human senses were introduced to assist program understanding and the response was to keep the term “visualization” by pointing out that it corresponds to the creation of mental models including all type of sensory modalities and not purely using visual objects. This claim was based on the last of seven definitions provided by [46], ignoring the previous six, that are more well known and generally adopted and also ignoring that the same source clearly states that the term is used with its “pictorial” meaning since the early 1880’s. We also have to mention that the roots of the word visualization can be traced back in Latin and are tightly coupled with the meaning of the English word “sight.”

Moreover the use of the term visualization in other areas as a pure “visual” concept raises the question whether the same term can have such a different meaning depending on the content.

With the existing terminology, phrases like “Software Visualization through sound, usually called program auralization [109]” should be accepted and the reader should accept that auralization is a sub-category of visualization.

Moreover, with the current terminology, the process to assist humans to get a better understanding of the software, using representations based solely on human’s vision is called Software Visualization (based on the first six definitions of the word visualization provided by [46]) while the process to assist humans to get a better understanding of the software, using multi-modal approaches, is also called Software Visualization, but this time based on the seventh definition provided by [46].

We propose a new term that reflects the current state of the field, encompasses all the existing tools and is broad enough to cover almost any type

of interface the future may bring. For us, what is now referred to as Software Visualization, should be called Software Noegenesis and should be defined as follows:

> Software Noegenesis is the process through which software is presented to humans, using appropriate interfaces, in order to amplify or assist the cognition of the software and its artifacts.

We selected the term Noegenesis, instead of apprehension, perception, conception, conceptualization, understanding, etc., because noegenesis is the result of noetic (mental, intellectual) activities such as perception, understanding, conception, thought processes, cognition, etc. [46]

Based on the new terminology and definition, in terms of modalities allowed, the area could be divided into Software Visualization, Software Auralization, etc., with Software Visualization having the meaning of representing software using only the visual sense, Software Auralization used for audible representations, etc., while also multi modal approaches are allowed after the combination of two or more of the sub-areas.

Although we propose this term, we will continue to use the traditional term of Software Visualization, in order to avoid confusions caused by the fact that according to the proposed definition, Software Visualization is a sub category of Software Noegenesis and not the whole area.

6. Taxonomy

In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that may be mutually exclusive, unambiguous, and taken together, include all possibilities. In practice, a good taxonomy should be simple, easy to remember, and easy to use.

Also we have to keep in mind that an important feature of a taxonomy is that it allows for expansion, if it is not already jointly exhaustive.

Having in mind the roots of the word, as described earlier, we define taxonomy as a jointly exhaustive classification according to a pre-determined system, used to provide a conceptual framework for discussion, analysis, or information retrieval.

7. Comments on Proposed Definitions

There are some consequences from the above definitions that should be mentioned:

First of all, a program can have various forms considering the way it can be specified (e.g., algorithmic description, source code description, machine language description, machine code, etc.), and different level of abstractions in each form that are form dependable. This flexibility is required in order to achieve a highly efficient visualization that will satisfy user needs.

Another consequence is that computer program may contain only the data definitions and not the data themselves that may be used as input or may be produced as output from the program. Data stored in databases that is known in advance that will be used from the program but not included in the program's source code, are considered a part of the software and not the program. The fact that during a program's execution there is an interaction between them and the program as a consequence of the instructions specified in the program, does not make them a part of the program itself.

A third observation is that, the result of the execution/interpretation of a program may not always be the desired one, hence giving the visualization the required width to also deal with "incomplete" programs which are program in an early stage of their creation phase (e.g. in the context of OOP, some methods may still be empty or with decreased functionality) or when executed do not have the desired behavior. The only assumption is that in all cases the program is assumed to be syntactically correct.

Consequently, from the definition of the program, the machine language representation of the program (the actual executable thing) is considered as a

different form of the program while the definition we accepted for source code is broad enough, compared to the definition previous researches are using or infer, without being bonded to any technological aspect.

Another important issue is the fact that according to our definition for Software Visualization, the input is software and any of its artifacts. This way we remove the restriction posed by Myers [28] that the input is a program written in a textual format. We need to remove this restriction because if textual programming ceases to exist sometime in the future, there will be no reason for Software Visualization. Another concept that can be derived from our definition for Software Visualization is that there is no restriction on the means that can be used to simulate the cognitive model to the user, in contrast with the definition provided in [50].

One significant consequence that we have to point out is the differentiation between program and algorithm, which is very subtle. Since an algorithm is actually a set of instructions and a computer is a device capable of quickly performing any given instructions, when those instructions are specified using a program, someone can easily realize the relation between those two concepts. This is the reason that in the area of Computer Science, a program is usually seen as an algorithm, expressed in terms understandable by a computer or as an abstraction of a program and algorithm animation was traditionally a part of the so called Program or Software Visualization.

We claim that an algorithm can exist and pose value on its own, without the need of a computer-related implementation since it can be implemented in many other ways and also because an algorithm exists in order to solve a specific problem (a computational problem in the case of Computer Science). In contrast, although a computer program is indeed a sequence of statements, it is not always an implementation of a specific predetermined well-known algorithm of the field of Computer Science, since there are many programs that simply do nothing, do not require any input, do not produce any output and do not manipulate any data or may implement algorithms that are outside the research

interest of Computer Science (e.g. the algorithm used to compute the monthly salary of an employer, or the tax a citizen has to pay).

Taking this into consideration, and also the fact that when Computer Scientists and the relevant books speak about algorithms, they actually refer to specific pre-defined computational problems and not to any algorithm that solves any (computational or not) problem, we separate the areas of Software Visualization and Algorithm Animation, providing only an overlapping area as shown in **Error! Reference source not found.**, where the size of the areas being unimportant. The overlapping area contains all the computer related implementations of the well-known predetermined algorithms such as sorting algorithms, etc.

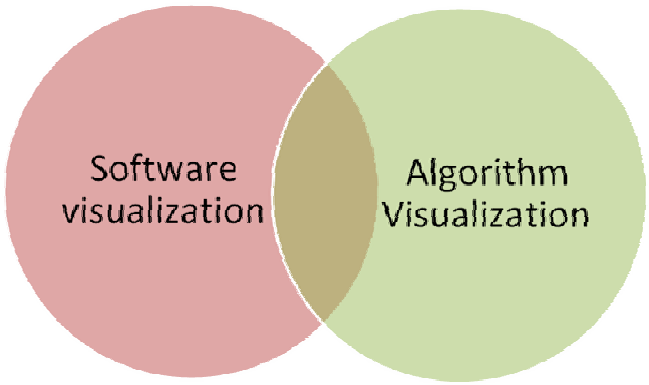


Figure 31 Proposed relation for the areas of Software Visualization and Algorithm Visualization

This way we do not restrict Algorithm Visualization in computer-related implementations. In other words, we consider Algorithm Visualization as a separate research area aiming to visualize algorithms (mainly for educational purposes). Talking about algorithms closely tied to Computer Science, the area of Algorithm Visualization may use “computer implementations” of those algorithms and try to visualize those implementations. In this case, the implementations should be treated as domain specific computer programs and

may be visualized using any appropriate technique borrowed either from the area of Algorithm Visualization or the area of Software Visualization.

On the other hand, in the case that a part (or even all) of a software contains an implementation of a well known computational algorithm (e.g. a sorting method) then this part may be visualized using any appropriate technique borrowed either from the area of Algorithm Visualization or the area of Software Visualization.

This kind of separation may seems to be an approach different from the widely accepted and traditional one that is defined by Price, Baecker & Small in [43] and can be shown in Figure 32.

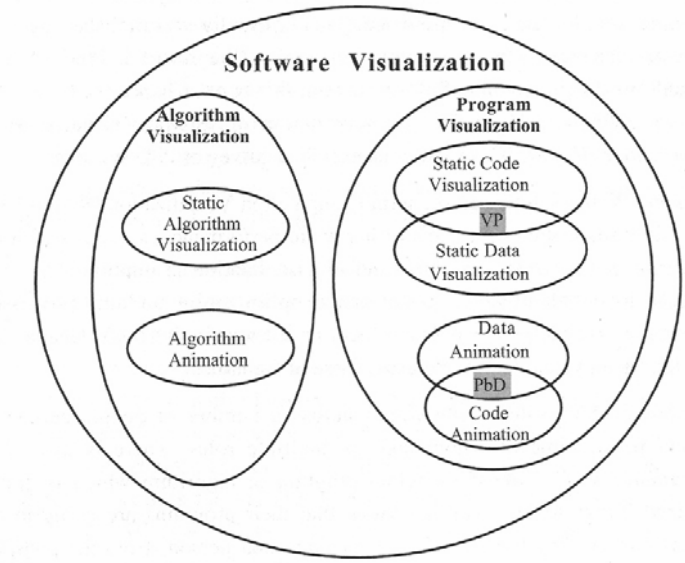


Figure 32 A Venn diagram showing the terms in the SV literature as proposed by Price, Beacker & Small [43].

On the other hand, our approach, is consistent with the view the same authors had in [50]:

> The differentiation between program and algorithm is subtle and can best be described from a user perspective: if the system is designed to educate the user about a general algorithm, it falls into the class of algorithm visualization. If, however, the system is

teaching the user about one particular implementation of an algorithm, it is more likely program visualization.

The last point is that we clearly differentiate Software Visualization from Visual Programming. According to our definition of Software Visualization, the term programming refers to a way to transfer the mental model that exist in the human's mind into a "mental" model understandable by the computer; simply just another form that software can take. In the area of Software Engineering, using a programming language is usually one of the last steps in this procedure, followed by the compilation of the source code that produces the desired binary form that is the "computer understandable mental model." There are other activities that take place before the pure programming, such as requirements analysis and software design. In that context, Programming (visual or not) and Software visualization are on the same path but differ in direction as can be seen in Figure 33.

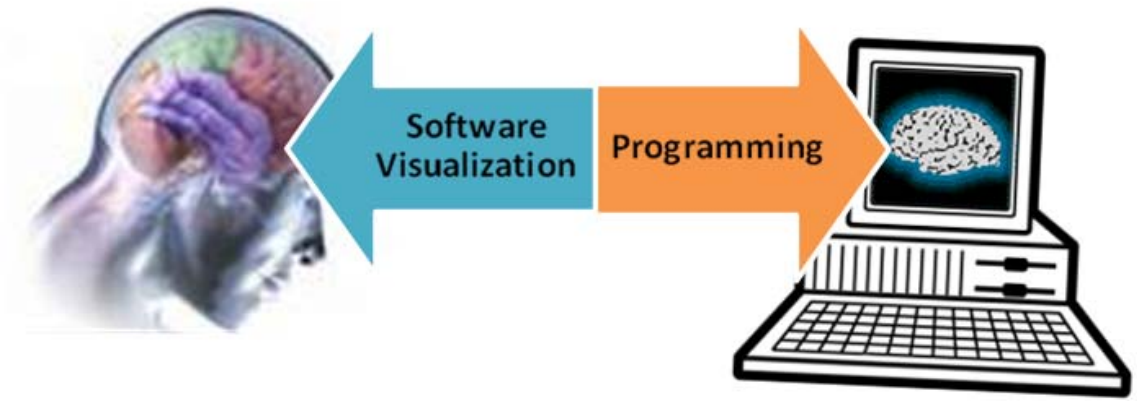


Figure 33 Programming (visual or not) and Software visualization are on the same path but differ in direction

This makes clear the line that distinguishes those two concepts and also shows how closely related they are. As a result of the fact that they both exist on the same path, they can share tools and "views." In other words, a specific representation of the software can be used both for programming and for

visualization purposes and its actual use separates the two views. For example, when a tools automatically creates a UML diagram given the source code, as an aid of software understanding, this action is part of the Software Visualization process while when we construct a UML diagram in order to specify a program and then we use a tool to automatically convert it into source code, this action is a part of Visual Programming.

C. FRAMEWORK DESCRIPTION

Based on the definitions we provided earlier, we see Software Visualization in a novel way: as an interface between software, in any of its possible forms, and the corresponding human's mental model for it, as shown in Figure 34.

Figure 34 Software Visualization as an interface between human and software

The two-sided arrows in Figure 34 represent only the interaction between the different parts during the process of Software Visualization and have nothing to do with the process of Visual Programming. This picture is derived after a top down analysis of Figure 33 and is actually one level of abstraction lower, representing only the Software Visualization part.

We intentionally didn't include any computer device or Software Visualization system in the above description. This way we describe the area in a flexible manner, allowing the software under visualization and the Visualization

System to co-exist in the same physical device or live in separate systems. This description also allows the Visualization system to be implemented in an all-in-one manner or with in a more modular way with its parts living in different machines. This is necessary in order to address current trends in Software Development that involve development teams being geographically separate, and also to encompass the future of Software Visualization which we believe involves the integration of Software Visualization engines into IDEs or elsewhere.

We divide the space “occupied” by Software Visualization in two main subspaces as can be seen in Figure 35, that are extracted observing the way we see Software Visualization; standing in between two distinct and significantly different entities, humans and software.

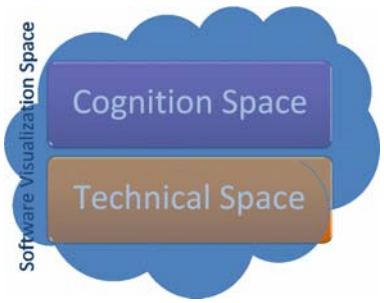


Figure 35 The spaces of Software Visualization, in the higher level of abstraction

1. Cognition Space

This level contains all of the concerns related with user’s communication with the visualization tools and the visualizations produced by them. It also concerns all the required concepts to address these problems. It concerns all principles and practices related to the cognition of software from human beings during the software development process and all the facts regarding the interaction with the user’s side, we should take into consideration during the development of Software Visualization tools. Knowledge contained in this space can be practically used to relate available representation to specific user tasks

and user purposes and to extract new visualization metaphors and representations.

This space contains questions like: “How do users understand software?”, “Who is the receiver of the visualization product?”, “What are the tasks he has to accomplish?”, “What are the intents of the user regarding a specific task?”, “What kind of interaction is required to assist the user to complete his task?”, “What techniques are available to increase the level of interaction and the level of understanding?”, “What are the principles that govern the communication with the user?” and many more.

2. Technical Space

This space concerns the required knowledge regarding the construction of the representations, the technical issues that are related with the data acquisition from the various software artifacts, the construction of the visualizations provided to the user and the construction of Software Visualization Tools. It encompasses all issues relevant to the implementation of the concepts of the cognition space. In order to construct a “view” for a software artifact we need data representing some properties of the software, transformed into a format more suitable for processing (if needed) and finally transformed into a representation, based on some pre-defined algorithms that define mappings between data and the basic building blocks of the representation. Here is the engine of a visualization. This space contains questions like: “What are the information encapsulated in the various software artifacts?”, “How can we extract information from the various software artifacts?”, “What other sources can we use to gain additional information for the software under visualization?”, “How can we integrate our visualization engine with other sources of data?”, “How can we integrate our visualization engine with other tools that support user’s tasks in a different way?”, “What problems arise during our communication with the software when we want to extract data for the visualization?”, “How can we solve the problems that we face during the communication with the software?”, “What kind of data are required to construct every representation?”, “How do we manipulate those

data?", "What techniques for data analysis should we use and what information do we expect to extract?" and many more.

3. Layers Overview

Our framework splits the area of Software Visualization into five that will also participate in the visualization pipeline, as we will explain later but not necessarily all of them and in the same order that they are presented here. An overview of the five layers can be seen in Figure 36 based on our view of Software Visualization as an interface between users and software.

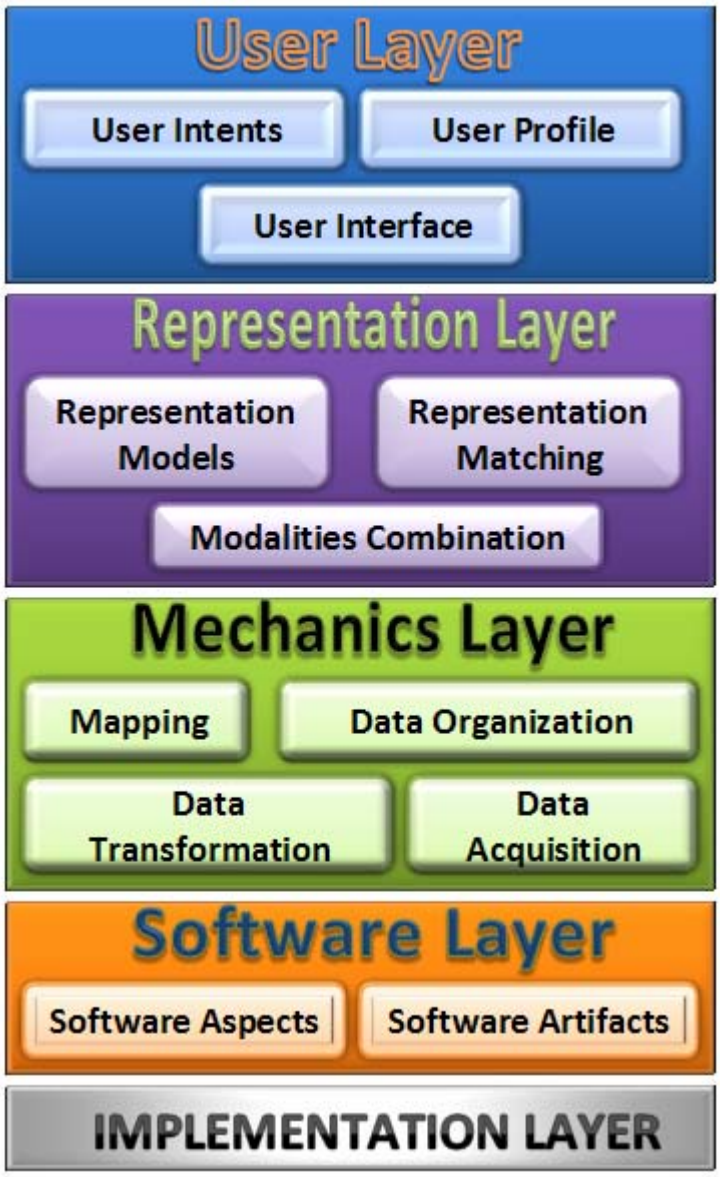


Figure 36 The five layers of Software Visualization

4. User Layer

This layer concerns all issues regarding the communications with the user, from the beginning of its interaction with the Software Visualization system when the user is phrasing a service request, until the end that he has finished using the system, looking to address usability issues, not only for the manipulation of the visualization product but for the communication with the system itself. This layer is defined by the *Who, What, Why and How* questions concerning the user; questions like “Why does the user need a visualization?”, “What information does the user expect to extract from the visualization presented?”, “How will the user interact?”, “Who will be the audience of our visualization and what are their special characteristics that we should take into consideration?” and many more. It’s obviously that significant answers for the questions of this layer should be scouted in related areas such as HCI, cognitive psychology, etc., and after the proper transformation, applied to our field.

Things that should be taken into consideration before the creation of any visualization are the following:

a. User Profile

Each user of a visualization is a unique entity and should be treated as such. One of the factors that will make Software Visualization tools more effective and help them escape from the academia space and become a part of a Software Engineer’s everyday life, is the attention they pay to match each individual user’s needs. This effort starts by trying to understand the users of our tools. Issues that we should take into consideration are:

- **User Models**: User modeling has been recognized, from other disciplines, as a very important and useful feature in several systems where the interaction with the user plays a crucial role. Concepts regarding the various user models, and how Software Visualization tools can be designed in a way that incorporates them, can be of great usefulness for our discipline, knowing that a big part of our tools are made to address the needs of a specific

audience. This way, flexible and profitable adaptations of the system behavior to the specific characteristics of the potential users of our tools can be made. Research on user modeling has been in a mature stage having made a lot of progress. The reader is referred to [110] - [114] as some examples of the applications of user modeling concepts in other disciplines, for existing classifications of this layer and how concepts from this area can help our discipline. Two very interesting attempts on mental modeling closely related to Software Visualization are made by Motta et al. [115] in an effort to model the behavior of an expert troubleshooter and by Storey et al. [116] where also a hierarchy of cognitive design elements are mentioned.

- **User Expertise**: As already stated by almost all the previous taxonomies of the area, one major concern should be whether the receptor of the visualization is a naïve programmer, an expert programmer, a student, someone that is familiar with the software aspect or artifact to be displayed, in terms of knowledge of its construction and its history (e.g. the programmer who wrote the source code) and in terms of specification issues concerning this artifact (e.g. displaying a UML diagram, perfectly laid out, ignoring the fact that he is not aware of all the details and symbols of this diagram may prove to be useless). As mentioned by Singer et al. [117] novices have very different behavior when interacting with software such as less focus since they do not have enough knowledge about what to look at.
- **User Role**: Another significant issue that governs the choice of the visualization to be presented in terms of what representation model will be used, what level of abstraction is the proper one, what details are to be visualized, etc., is the role of the user in the software development process. For example, if the project manager

and a programmer want to know about the evolution of the software in a project, different level of details will be required, probably forcing different representation models to be used.

b. User Intents

Many of the existing taxonomies, amplify the importance of matching the user needs relative to his intents when appealing to a visualization system, but user intents have either been limited to traditional programming tasks (e.g. debugging, execution traces, data structures, etc.), or to pedagogical reasons or scattered along different parts of the taxonomy. Even the latest taxonomy proposed by Maletic et al., is limiting user needs to traditional Software Engineering tasks, not providing a complete answer to the question “Why is the visualization needed?” and also not answering questions like “What are the user’s specific expectations and goals when he requests the visualization?”, “What are the user’s needs that the representation should fulfill?”, etc. User intention is a crucial issue, since it represents the reason a visualization is requested, and cannot be described simply by a generic description of the Engineering task to be accomplished. In order to fully determine user intention, the following issues should be taken under consideration:

- **User Engineering Tasks**: Issues related to the tasks that software engineers have to accomplish during a software’s lifecycle, is a very significant parameter in the process of determining the proper visualization. In order to establish a common base for related issues, we propose a division by grouping tasks based on the major areas of Software Engineering as they are defined by The IEEE’s Software Engineering Body of Knowledge [118], such as Software Requirements tasks, Software Design tasks, Software Construction tasks, Software Testing tasks, Software Maintenance tasks, Software Configuration Management tasks, Software Engineering Management tasks, Software Process tasks, and Software Quality tasks. A comprehensive list of the subtasks associated with each

group and a further subdivision of them, is out of the scope of this paper since it will not further assist the presentation of the framework. Something that we want to point out is that the above classification of tasks provides the necessary space to our field to include areas such as the visualization of software metrics or the visualization of Software Evolution. These are two examples of software lifecycle activities, whose products or tasks are not covered from previous taxonomies, even if those important activities have their own representatives in the library of Software Visualization tools (e.g SeeSoft. GEVOL, EPoSee). The only exception is the taxonomy proposed by Maletic et al., even though the inclusion of these areas is done in a generic way.

- **User Goals**: Each user of a Software Visualization system, no matter what task he wants to accomplish, has some goals that differ each time he requests a specific visualization. Those goals should be clearly identified and supported by the tools. User's goals, according to Bergeron [119] can be broadly classified as analytical (user knows what he is looking for and the visualization will help him determine whether it is there and where exactly), exploratory (user does not know what he is looking for and he expects the visualization to assist him to understand the nature of the object visualized) and descriptive (the phenomena visualized are known and the user needs to present a clear verification of this phenomena). A further analysis will reveal operations at a more detailed level such as search, browsing, comparison, learn, understand, monitor, etc., revealing how viewer's goals differ for the same representation. One of the most comprehensive classifications of user goals in terms of actions needed to be accomplished, is the one provided by Wehrend & Lewis in [120] that proposes a number of operation classes (identify, locate,

distinguish, categorize, cluster, distribution, rank, compare, within relations, between relations, associate and correlate). As a final comment, we would like to say that we wouldn't be surprised if, in the near future, we see logical task description languages, similar to the approach proposed by Casner [121] for scientific visualization.

c. User Interface

In order to fulfill each individual user's needs, attention should be paid to the interface that the user is exposed to. The ideals and the required properties for user interfaces along with the techniques used to facilitate the use of the system and make the communication with the system and its visualization products more effective, are concerns of great importance. All the existing taxonomies mention some aspects of the required properties and techniques for the interface of the visualization presented and the techniques we can use to make it more usable and effective, but they are mentioned in a scattered way with the exception of the Maletic et al. taxonomy. Moreover, no taxonomy mentions that the interface of the Software Visualization Tool itself is also under judgment and should comply with the results of the Human Computer Interaction discipline. It's not only important what facilities a tool is offering but also how well it offers them. A third dimension, mentioned only in some of the existing taxonomies, is the interface provided by the Visualization System when direct access to the software artifact is requested from the user or is required for the production of the visualization.

Concepts that govern the use of the system and its interface during the initial phase where the user phrases a request until the time he will stop using the system can be split into three groups:

(1) **Software Visualization Tool Interface.** All concerns regarding an HCI approach for the Software Visualization tools themselves should be considered. There are things not yet considered by previous taxonomies such as the ability to store the created visualizations, their future

retrieval and automatic update, the ability to group and characterize user created facilities (similar to the “favorites” concepts of web browsers) or the level (and ability) of integration of the Software Visualization tool with other everyday tools used by software engineers, the level of the tool’s adaptability to specific users or even whether they provide mechanisms to authenticate users, control access to data and log identity of those accessing data (something that the majority of real life’s Software Engineering tools have as a standard) and many more. Another significant issue is the interface that tools are providing in cases when the user has to interact with the artifacts themselves (e.g. the user is instrumenting the source code on his own). There are cases that Software Visualization tools do not provide an interface for the artifact itself, forcing the user to use other applications in parallel. There are so many examples out there that we can adopt; like the example of the browsers that display an “artifact” (e.g. a pdf file) incorporating the interface of a specialized editor.

(2) **Visualization’s Interface.** Many things have been said and written about the ideals and the techniques addressing the concerns of this layer. There exist principles and techniques that govern user navigation issues; user interaction issues; principles related to the cognitive economy of the final representation; proper presentation and reproduction of the presented visualization in terms of effectiveness such as consistency, simplicity, clarity, understandability, visualization economy, scalability, and much more. There are a number of techniques to increase the effective communication with the user from zooming techniques to fish eye views and from controlled distortive views to focus and context techniques, just to name a few, whose use is supposed to assist interaction with the visualization presented to the user. Despite all these, and due to the fact that no special focus have been given to the concerns of this layer, cases where pictures are used in situations where text would be more economical, exaggerating the “intuitive nature” of graphics, are not so rare. The main issues that should be take into consideration are:

- **Interaction**: The main issue is the degree of interaction between the user and the visualization. There are a number of methods & techniques used to improve the interactivity of a visualization. There are many descriptions for user needs during his interaction with a visualization, from the ones falling into the sphere of ideals, such as Schneiderman's [122] proposal (overview-zoom-filter-details on demand-relate-history-extract), up to more practical ones, such as the Wiss et al. [123] proposal (which expands Schneiderman's categories at a lower level) or the approach proposed by Zhou & Feiner [124]. Also here are all other techniques such as contextual zoom, focus + content, the various implementations of the fisheye view, dynamic queuing using slides and many more that are used to realize all the user needs while he is interacting with a visualization. Mixing this knowledge with the classification of user goals, explained in the previous layer, a matching between user goals and the applicable technique is possible.

- **Lay out & Aesthetics**: All concerns regarding the principles and practices on laying out the visualization primitives and how we order parts to create the whole, issues regarding the visualization frame such as the various graph layout algorithms that may be used, the principles and techniques for color & texture use that will facilitate our goal and will not cause a cluttered visualization, whether the objects placed on the screen are on a size that conveys information, the use of multiple views, etc. Issues like the smoothness of animations produced are also placed here since we consider animation as a visual discourse[14] and as such the various displays are combined to produce the whole.

[14] The term is used in the way it is defined by Zhou & Feiner in [126] to denote a series of connected displays or a cohesive formation of a series of visual frames that will produce an animation.

- **Medium**: The medium that the visualizations are going to be presented enforces a significant number of constraints revealing research issues that need further attention. Presenting a diagram on a 17” monitor is quite different from presenting the same diagram on paper or an array of screens. It’s obvious that each medium has its own attributes and properties that should be taken into account when a visualization is created making the concerns in this layer a very significant one for the final result.

5. Representations Layer

The ultimate goal of a Software Visualization tool is to match user profile and intents to a specific set of representations; this is the only way to make a visualization tool useful and desirable. The importance of this task, throughout the visualization process, was realized very early on, as we can see in a classical book by D.A. Norman where the author states: “it is possible to determine the relationships between actions and results, between the controls and their effects, and between the system state and what is visible [125],” and Eisenstadt et al. [68] that state that “A major research theme common to the different possible modes and styles of software design is that of finding a (graphical) representation for program behavior which provides a good mapping to the way the programmers themselves tend to formulate solutions,” just to name a few.

Since, after all, everything is about modeling, this layer contains all the concerns regarding available (or proposed) models we use to visualize software, the metaphors they are based on and their properties, the information that are conveyed from each of them, their advantages and disadvantages and the visual structures and representation they use along with the degree of the multi-modal approach applied to further assist the matching process.

More important it addresses the problem of matching user profile and intents to a specific set of representations containing all the questions related to the options we have in our effort to transform the non-spatial entity of software into something, more or less, closer to human cognition. In other words, this layer

is defined by the *What, Why and How* questions concerning the representations that will be presented to the user having determined his profile and intents; questions like “How should we represent for the specific user and his intents?”, “What information can be conveyed by a visualization?”, “How can different modalities be combined to increase the information conveyed?” and many more. Concerns that deal with this layer include:

a. Representation Models

In our area, there have been efforts for a systematic approach, for classifying existing representations, like the one proposed by Brown [80] for algorithm animation displays, the classification proposed by Roman & Cox [51] and the one by Price et al. [50] while areas like Information Visualization and Scientific Visualization have a long history on classifying the available visualizations.

One of the most noticeable proposals is the classification of the visual hierarchy proposed by Zhou and Feiner [126] that can be seen in Figure 37. A classification that is also applicable for all kind of modalities.

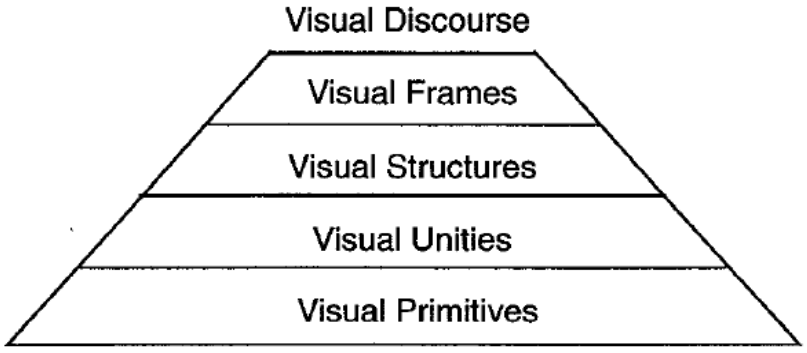


Figure 37 The Visual Hierarchy proposed by Zhou & Feiner [126]

Each of the levels in the hierarchy can be subdivided based on the contents of each layer in the hierarchy. For example, the visual[15] structures that

[15] The term visual here is used its basic meaning connected to human’s vision.

are available for representation of the various software artifacts, can be classified at a first level based on the existing approaches into seven categories:

- Text
- Tables
- Charts
- Graphs
- Diagrams
- Maps
- Cartograms

Each of the above categories, contains a number of available structures (e.g. for the diagrams category we know there exist flowchart diagrams, Nassi & Schneiderman diagrams, the various UML diagrams, etc.), but, most important, all the properties for each, such as its level of abstraction, its applicability for different kinds of data[16] (which in our case will be the various software aspects that we will explain later), the underlying data model required for construction, its properties such as level of abstraction, its dynamics, etc.

We believe that a future revisit of these issues is required, so that a complete taxonomy is created along with a clarification of the properties for each element and maybe an empirical evaluation for each of them.

b. Representation Matching

For a given user profile and intents, the challenge is to match them with the proper representations. Here we encompass all concerns that address this problem. In his classical paper [127], MacKinlay defines two criteria for this activity, *expressiveness* and *effectiveness* with the former being a criterion of "the capability of the metaphor to visually represent all the information we desire to visualize [73]," and the latter related to "the efficacy of the metaphor as a mean of

[16] An interesting approach that correlates each visual structure with various data can be found in [129].

representing the information [73].” Roth and Mattis [130] add another criterion for the appropriateness of a representation: the user’s goals in viewing the representation; a view that is missing from previous taxonomies. Eisenstadt et al. [68] name this ultimate goal the “goodness of mapping.”

These should be the driving forces during research conducted for the matching of user needs to specific visualizations. We believe that “know how” from relevant disciplines can be easily applied to our discipline along with research for empirical evaluation regarding the various matching solutions.

As a final comment, we have to mention that this matching can occur in a fully automatic way (the SV system will decide which visualization is the proper one), or in a semi-automatic way (the system will propose and the user will choose whatever he thinks is the fittest) and manually (in systems that the visualization is manually defined and constructed by the user, where the user clearly decides what he believe is most appropriate for him).

c. Modalities Combination

One of the main characteristics of every visualization presented to the user is which of the user’s senses it stimulates. Traditionally, the area was looking at stimulating only the vision of the users, but other approaches have emerged, taking advantage of other senses. This layer concerns the issues that arise when combining models that use different modalities to achieve a multi-modal product.

6. Mechanics Layer

This layer concerns the required knowledge regarding the construction of the representations and the technical issues that are related to the data acquisition form the various software artifacts. This layer is defined by the *What* and *How* questions concerning the creation of the representations: questions like “How will we map the visualizations primitives with the existing data and vice versa?”, “What are the available data sources?”, “How should we extract information from the various software artifacts?”, “How will we interact with the software under visualization?”, “How can we integrate our visualization engine

with other sources of data?", "How can we integrate our visualization engine with other tools that support user's tasks in a different way?", "What problems arise during our communication with the software when we want to extract data for the visualization?", "What kind of data are required to construct every representation?", "How do we manipulate the data we have gathered from the various sources?", and many more. It's obvious, that in order to answer some of the above questions, knowledge from related research areas should be extracted and after being properly transformed, applied to our discipline.

This layer contains all concerns relevant to the implementation issues of the ideals defined in the upper level. In order to construct a "view" for a software artifact we need data representing some properties of the software, transform them in a format more suitable for processing (if needed) and transform them into a representation, based on some pre-defined algorithms that define mappings between data and the basic building blocks of the representation. Here is the engine of a visualization system and the part of our domain that deals with the mapping of the non-spatial data we have from the lower layer to a visual form using the proper models and abstractions. Issues contained in this layer can be grouped as follows:

a. Mapping

Mapping data to visual elements of the representations and actually constructing and rendering them is a rather complex activity. Primitive graphical objects (boxes, lines, circles, colors) or primitive audible objects (sounds, duration, tone, etc.) must be defined from the existing data and further combined to create a useful view. Specifications required for the creation of the corresponding representation model, must be known and well defined, e.g. if the user asks for a UML type diagram to observe the inheritance properties for a number of classes, knowledge related to the symbology of UML is required. This group deals with principles, practices and techniques used for transforming data into a representational framework (schema)[17] that will be used for the

[17] As defined by Russel et al. [128].

construction of a series of visual frames to be presented to the user. The other side of mapping is the transformation of user's requests into data required to be extracted from the software under analysis.

b. Data Acquisition

Here we deal with all problems faced during our effort to extract useful information from the various software artifacts or other sources along with all issues concerning the communication of the Software Visualization systems, with the various sources of data (e.g. software artifacts that exist in the same physical location with the Visualization system or in a different geographical place). Techniques like post-mortem extraction, source code instrumentation, run-time extraction, etc., as well as their properties in terms of invasiveness, type and amount of information they provide, etc., are only some of the well known techniques, each of them with its own merits and appropriate for specific usages. Data that are acquired from the various sources are considered to be in a raw format, in the context that they are in the format provided by their source.

c. Data Transformation

The enrichment, analysis and transformation of the acquired data are activities of great importance in the creation of any visualization. Enrichment may contain activities from simple ones like time stamping up to more complex one like filtering. Analysis refers to activities such as metrics computation (if not provided by a data source), cross analysis for the data obtained from the various software artifacts (e.g. the combined analysis of data extracted from the execution of a program with data from a static analysis, etc.), while transformation deals with the conversion of the acquired data to a format that is more appropriate for the Visualization system. Other issues that should be taken into consideration are the concepts & principles regarding automated static software analysis, data hierarchical analysis, metadata extraction, data pattern recognition, etc.

- **Data Organization**

Data acquired from the various sources should be spread along the various modules of the Visualization System on a per-demand or constant flow basis in order to be used for the creation of the visualizations. Concerns like the use of a bus-type architecture or a client-server architecture for the management of data along with their pros & cons, are included in this layer along with relevant concepts from knowledge organization and structuring.

7. Software Layer

As we have already mentioned, software can take various forms and can be seen with different views, each of them related to specific representations, from source code in its textual form (the most typical case) and bits and bytes to executable files up to higher level UML diagrams or plain text software documentation. This layer is defined by the *What,* and *How* questions concerning the software under visualization: questions like “What software aspect are we going to visualize?”, “How each software artifact participates in each distinct software aspect?”, “What are the information encapsulated in the various software artifacts?”, “What level of abstraction is each software artifact providing?”, and many more. The issues addressed by this layer can be grouped as follows:

a. Software Aspects

We can categorize the various aspects based on their dynamics and their object of observation. There are the static aspects of software under visualization such as its structure, its components, its metrics, etc. These aspects can be extracted without the need for observing the behavior of the software but concerns all those properties that are inherent in the software when it is resting in a repository.

On the other hand, there are software aspects that require the software to be “alive” and executing and actually refers to its runtime behavior such as the real-time control or data flow, the dynamic call chain (we mainly refer to OOP where the dynamic allocation of objects make things more

unpredictable), the call hierarchy, the hardware utilization issues, the various views of causal relations, software's behavior during test cases or debugging, etc.

The declaration of the software aspect to be presented to the user is one of the primary concerns that should be clearly defined form the early stages of the visualization process, since it has a major effect on how things will be presented and what data need to be extracted.

b. Software Artifacts

The main source of data are the real software artifacts, containing a plethora of information that are sometimes unutilized and unexploited. All these sources should be categorized and carefully analyzed in the context of Software Visualization concerns, so as to discover and classify the information that they can provide combined with the software aspects the user requested to be visualized. These can be subdivided into three groups:

(1) **Hardware Artifacts.** All artifacts that relate to the hardware of the Software System we want to visualize belong to this layer. The system's platform, operating system, memory, number of processors, resources sharing policy, particular processor characteristics, quality of service for network links, and requirements for co-location of components to ensure performance and others exist in this layer. Caution should be taken that this layer contains only the facts for the hardware issues of the Software System under visualization. How those facts are affecting the data extraction phase, and how they impose restrictions on this process, is addressed by the appropriate sub-layer of the Data Layer defined above.

(2) **Software Artifacts.** All artifacts that relate to the software under visualization and the forms in which it is available along with any information that can be extracted by them, our ability to transform them from one form to the other, etc., are contained in this layer. This layer can be further

subdivided, based on the classification proposed by Koschke [131] in the Architecture Layer, Middle Layer and Lower Layer, each of them containing the corresponding information.

8. Implementation Layer

When it's time to create a Software Visualization tool, there are decisions that we should take and trade off to balance, as in any other software. Issues such as the architecture & design of Visualization systems and their pros & cons, the resource allocation problem for the implementation phase on a Visualization System, issues concerning the specification of work flow inside Visualization Systems and how it can be optimized with respect to the resources available, how to enact the work at the required time, programming paradigms that may be used for the construction of Visualization Systems, instruction execution rates, cache utilization, processor utilization, concerns regarding operating system level activities, system calls, address space activity, communication library level activity and many more.

D. VISUALIZATION CYCLE – VISUALIZATION PIPELINE – REFERENCE MODELS

Having accepted that Visualization can be seen as an interactive communication between human and computer, with the Software Visualization system being the interface for the communication of the two parts, we have to define the visualization process.

Duke et al., in [132] present a visualization cycle that embodies the roles of human and the system (in our case the Software Visualization system and the software) that can be seen in Figure 38.

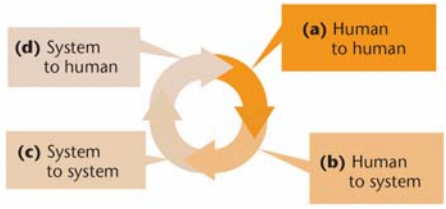


Figure 38 Visualization cycle proposed by Duke et al. [132]

In this diagram we see all the activities that take place during a Visualization, and as noted in [133] it‘s better to imagine this cycle as a spiral expanding out in time. More specifically, the various phases are:

- human to human, dialogue between domain and/or visualization experts to explore the problem requirements
- human to system, data to be visualized, required representation, and/or the process to be used
- system to system, specification of services including data models and functional behavior
- system to human, visualization product output to user for inspection

It is obvious that in our case, we address the steps (b) & (c) & (d) but we may need to take into account that Software Visualization tools should also support the activity described in (a); a concept that does not exist in any of the previous taxonomies / frameworks of the area but can be addressed in our User Interface Layer and Implementation Layer.

Now that we have an abstract framework for the description of the activities that take place during a visualization process, we need to zoom in and explore the visualization process itself, by establishing a reference model that will describe it in a less abstract but high level, We have to remember that, the purpose of a reference model is to specify the internal steps of the visualization process and not the technologies or the implemented components that participate in this process.

One of the most widely accepted reference models for the visualization process is the one that Haber and McNabb [134] proposed. It is a dataflow oriented reference model that describes the visualization process as "a specific sequence of data enrichment and enhancement transformations, visualization mappings and rendering transformations that produce an abstract display of a scientific data set." The reference model they propose can be seen in Figure 39.

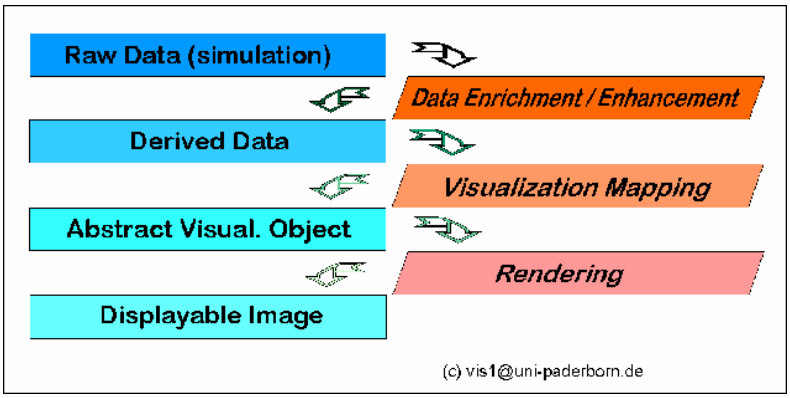


Figure 39 Haber & McNabb's reference model [134].

The model proposed by Oudshoorn et al. [53] is shown in Figure 40 and is also a one-way model, mainly describing traditional types of Software Visualization process.

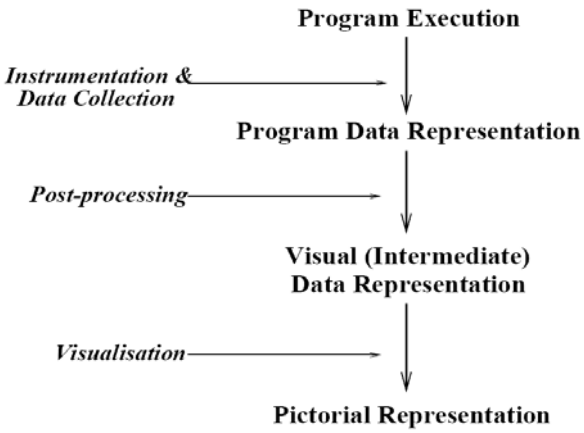


Figure 40 Oudshoorn's transformation series to produce program visualization [53].

This model is closely related to the model proposed by Card et al.[74], which can be seen in Figure 41. This model is been used as a basis for the taxonomy proposed by Maletic et al. [73].

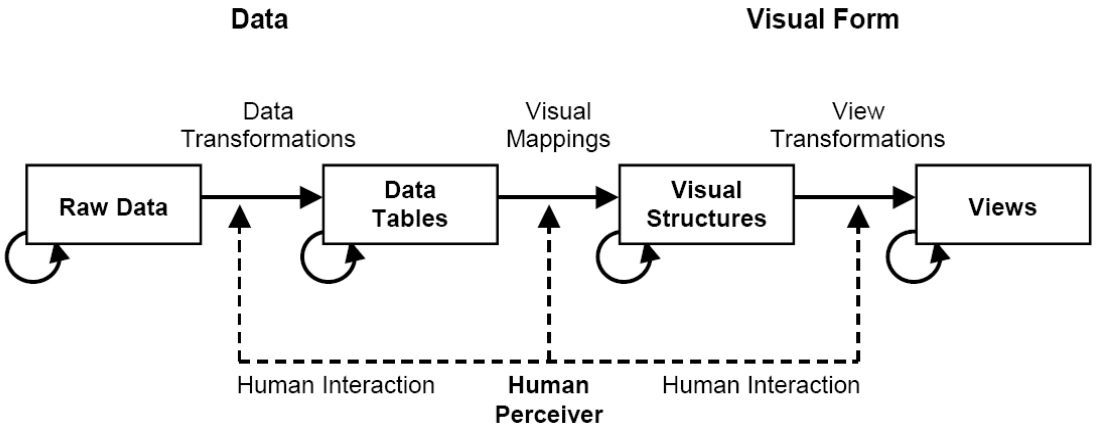


Figure 41 Card's Reference Model [74].

Another interesting reference model has been proposed by P. Robertson and L. De Ferrari [135] and is shown in Figure 42.

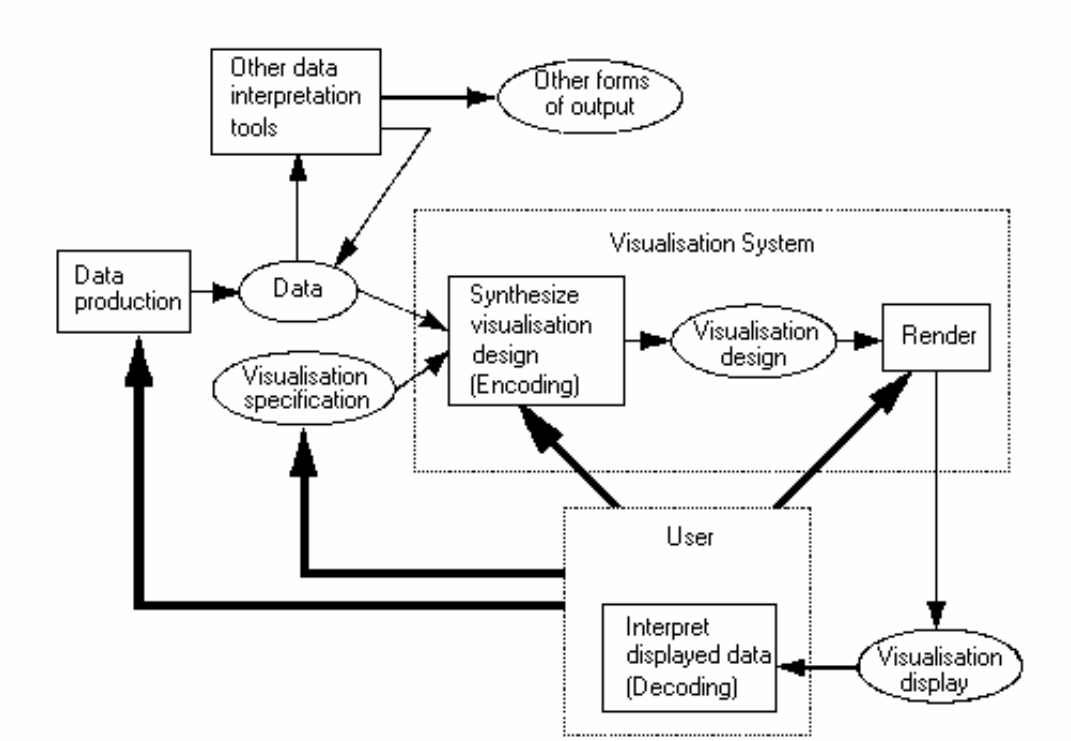


Figure 42 Visualization Reference Model proposed by Robertson & De Ferrari [135]

Finally, a multi-layered model has been proposed from Brodlie et al. [136] and is shown in Figure 43. This model concerns the special needs of Scientific Visualization over a grid of resources and is based in the Haber and MacNabb reference model.

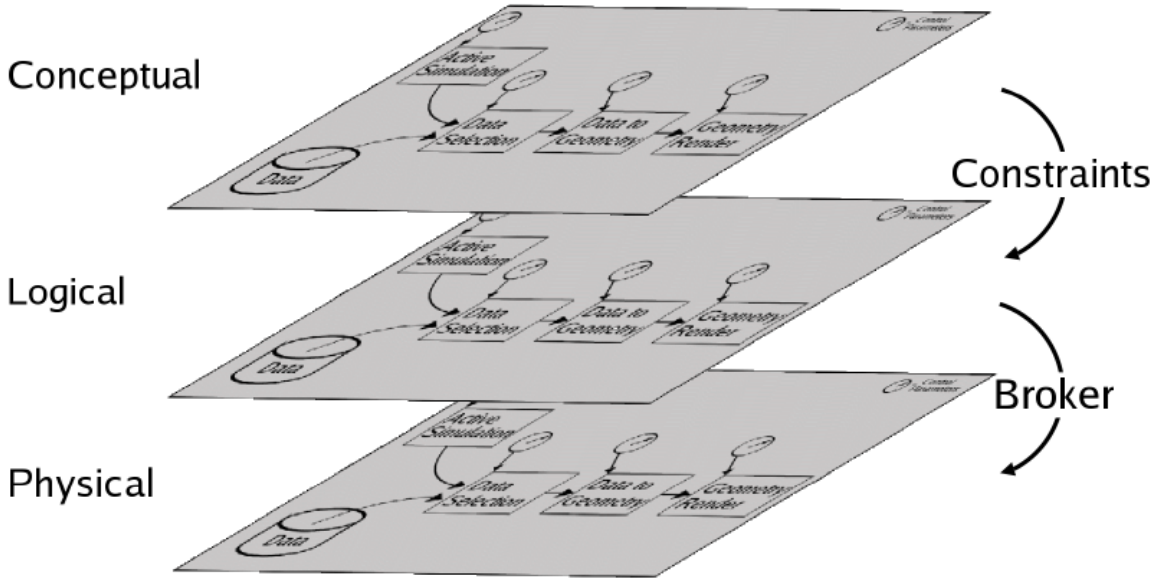


Figure 43 Reference Model proposed by Brodlie et al. [136]

The reference models mentioned above that describe the visualization process are either a one-way visualization pipeline or are not applicable in our case and hence we represent the visualization pipeline as a clearly two-way process involving the three parties of our view of Software Visualization as an interface, having also matched the corresponding activities from the visualization cycle, as can be seen in Figure 44. Our model has similarities to the one proposed by P. Robertson and L. De Ferrari, with the major differences being the fact that we concern the data acquisition process a part of the Software Visualization system and we include communication concerns with the various data sources.

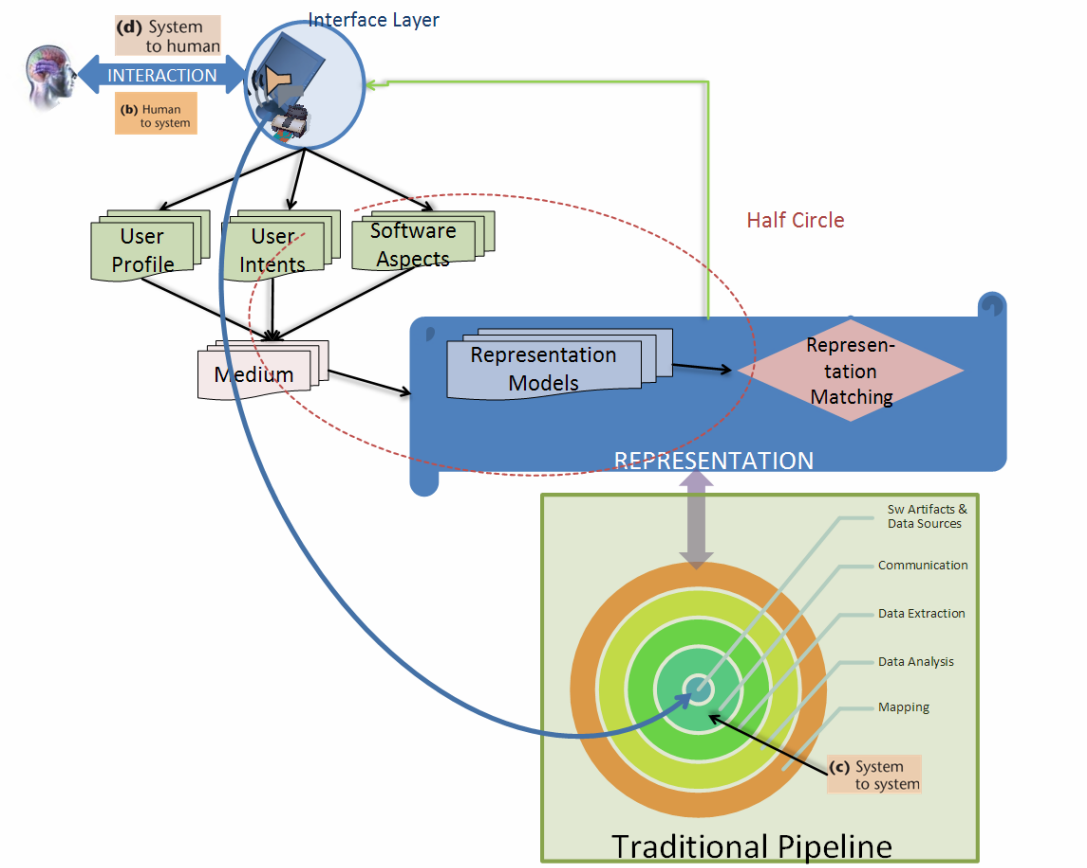


Figure 44 The Software Visualization process as a two way communication

The proposed visualization cycle is also based on our initial view of Software Visualization as an interface between human and software artifacts. In order to provide a better description of the proposed visualization cycle, we will provide two, rather extreme, examples:

- The cycle starts with the system interacting with the user to obtain information regarding his profile and intents and then the user specifying his needs rather than by specifying a particular visualization. Knowing the characteristics of the user, the available medium, the software aspects that will participate and the available representations, the appropriate set of visualizations is defined. Then these visualizations are mapped to the required data

instructions, are defined and data are extracted form the software artifacts or other data sources. The data obtained are processed and then mapped to the primitive elements of the representation, rendered and presented to the user, which in turn interacts with them. Another cycle may be started by a simple “zoom” request by the user; a request that may be fulfilled by a “half cycle” (red dotted line) if the data required to construct the new visualization are present. In case the interaction is something more complex (like a completely new set of visualization, an additional view, etc.), this will cause a new full cycle to initiate.

- The cycle starts with the user starting to instrument the artifact himself, hence all the layers are bypassed (blue solid line) since the user has decided about his needs, he has decided on his own what kind of views will fulfill his needs and he did the mapping of those views to the data that are required to be gathered. After completion of his job, all that is left to be done form the Software Visualization system, is to map the data obtained from the artifact to specific primitive elements (which can also be done manually by the user) and display the produced visualization to the user.

E. CONCLUSIONS AND FUTURE WORK

It is indisputable that Software Visualization is not a part of everyday practice for the majority of people involved in Software Engineering. We believe that the best way to overcome this reality is to dig deep, reach the roots of our discipline, face the reality and reframe our research in a way that will enforce better cooperation with other research areas and better transfer of “know how.”

In this effort, we proposed a new name for our discipline that clears any ambiguities that may exist. We also proposed a new framework based on the view of Software Visualization as an interface in between human cognition and the abstract world of Software.

We believe, that our approach breaks the bonds of the traditional concepts of Software Visualization that kept it loosely coupled with the Software Engineering reality, and defines a new framework that will allow for the construction of new types of Software Visualization tools. We propose a flexible approach, easily expandable that can adopt new principles, practices and technologies. Splitting the area into layers, we have a taxonomy that is mutual exclusive and complete. We have an ordered separation of the issues faced and we gather all relevant knowledge in the same place. Our framework can serve as a map for a newcomer to the area, as a body of knowledge and as a guide for research and, on the other hand, as a concrete framework for the relation with other disciplines and knowledge exchange. The way we laid out the layers is close to the visualization pipeline to simulate the creation of new multi-modular types of architecture for Software Visualization tools.

As clearly stated by Brooks, there is no “silver bullet” that will solve all of our problems regarding the complexity of software. Knowing that, we see Software Visualization as another tool in our hands. What we propose is that we should change the way Software Visualization systems are made. We should not expect a single tool to answer all the questions but for an integrated set of tools that can answer a larger number of questions.

There are many specialized tools that implement some of the required functionality, but usually in a standalone fashion. Based on the layered approach we expect to see architectures that delegate the various concerns described in the different layers to existing tools and finally integrate the visualization engine into existing, broadly accepted tools to the Software Engineering community. Having many different tools in a toolbox is not always the best solution and this is one of the reasons that people do not accept the majority of the tools produced

so far. The answer to this is integration. With the proposed layered framework for our discipline, this is clearer, since each specialized tool can find its own place in one or more layers.

Another benefit of the proposed multi-layered approach is that it creates the vital spaces for new technologies to be integrated with existing ones. All kind of representations and sensory modalities are allowed; from the traditional textual to pictorial, aural or even the transmission of electromagnetic waves using a brainwear device, is assumed to be another form that a part of the software can take.

We do not claim to be the first evangelists of modularity as the future of the field. Nielson et al. [137], many years ago claimed that modularity in design is necessary if we want to create visualization systems that are really useful for a wide range of users that gives us a wider range of tasks. A statement made from a researcher in the area of Scientific Visualization that holds equally to our field.

Another significant issue that presents great importance, both for the research society but also for the Software Engineering practitioners, is the area of Software Visualization tools evaluation. We believe that if our framework is used toward this direction both a quantitative and qualitative approach for all concerns may be achieved. A full expansion of our multi-layered model, can serve as a quantitative model for evaluating tools, with the same labeling concepts proposed by Price et al. [50], and at the same time can serve as a guide for a qualitative analysis of each of these areas of concerns.

It’s clear that before this is achieved we have to agree on some standards and this is another interesting area for future work. Standards on the classification of the existing representations and the information that can be conveyed from each of them, standards regarding the communication protocols between the different modules or applications and many more.

Of course, we know that this approach is a means to an end and not an end to itself. A future revisit of the proposed layers is required, in order to provide a more detailed description of the concepts & principles including in each of them.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Wirth, N. (1971). Program Development by Stepwise Refinement, Communications of the ACM 14(4), April 1971, 221-227.

2. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. (1972). Structured Programming. Academic Press.

3. Parnas D.L., "On the Criteria To Be Used in Decomposing Systems Into Modules," Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-1058.

4. Booch Grady, Object-Oriented Design with Application. Benjamin/Cummings, 1991. ISBN: 0-8053-0091-0.

5. Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1998.

6. Kant's Immanuel, Critique of Pure Reason. In Commemoration of the Centenary of its First Publication. Translated into English by F. Max Mueller (2nd revised ed.) (New York: Macmillan, 1922) p. 41.

7. Poilâne Lionel & Poilâne Apollonia, Le pain par Poilâne -. - Paris : le Cherche Midi, DL 2005 - 1 vol. (389 p.) - ISBN 2-7491-0143-3 (rel.): 35 EUR. - EAN 9782749101439.

8. Koschke R.. Software visualization in software maintenance, reverse engineering, and reengineering: A research survey. Journal on Software Maintenance and Evolution, 15(2):87--109, 2003.

9. Bassil S. and Keller R., Software Visualization Tools: Survey and Analysis. In 9 th International Workshop on Program Comprehension (IWPC 2001).

10. Goldstein, H. H. & von Neumann, J. (1947). Planning and Coding Problems of an Electronic Computing Instrument. In A. H. Taub (Eds.), von Neumann, J., Collected Works (pp. 80-151). New York: McMillan

11. Nassi, I. and Shneiderman, B. 1973. Flowchart techniques for structured programming. SIGPLAN Not. 8, 8 (August 1973).

12. Haibt, L. M. (1959). A Program to Draw Multi-Level Flow Charts. In Proceedings of The Western Joint Computer Conference, 15 (pp. 131-137). San Francisco, CA:.

13. Knuth, D. E. 1963. Computer-drawn flowcharts. Commun. ACM 6, 9 (Sep. 1963), 555-563

14. Roy P. and St-Denis R., Linear Flowchart Generator For a Structured Language, SIGPLAN Notices (ACM) 11, 11, (Nov. 1976), 58--64.

15. Naur, P. (Ed.) (1963). Revised Report on the Algorithmic Language ALGOL 60. Communications of the ACM 6(1), 1-17.

16. Conroy, K. and Smith, R.G. (1970). NEATER2: A PL/I Source Statement Reformatter, Communications of the ACM 13, 669-675.

17. Hueras, J. and Ledgard, H. (1977). An Automatic Formatting Program for Pascal. SIGPLAN Notices 12(7), 82-84.

18. Knuth D., “Literate Programming”, The Computer Journal, Vol 2, No , 1984,pp97-111.

19. Baecker, R.M. and Marcus, A. (1990). Human Factors and Typography for More Readable Programs. ACM Press, Addison-Wesley.

20. Knowlton, K. C. (1966). L[6]: Part II. An Example of L[6] Programming. 16 mm black and white sound film, 30 minutes. Murray Hill, NJ: Technical Information Libraries, Bell Laboratories, Inc.

21. Baecker, R.M. (1981): With the assistance of Dave Sherman, Sorting out Sorting, 30 minute colour sound film, Dynamic Graphics Project, University of Toronto, 1981. (Excerpted and 'reprinted' in SIGGRAPH Video Review 7, 1983.) (Distributed by Morgan Kaufmann, Publishers.).

22. Baecker, R.M. (1968). Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures, Prof. First Hawaii International Conference on the System Sciences, January, 1968, 128-129.

23. Booth, K.S. (1975). PQ Trees, 12-minute color silent film.

24. Lieberman, H. (1984). Seeing What Your Programs Are Doing, International Journal of Man-Machine Studies 21(4), October 1984, 311-331.

25. Brown, M.H. and Sedgewick, R. (1984b). A System for Algorithm Animation. Computer Graphics 18(3), 177-186.

26. Ronald Baecker and Blaine Price. The Early History of Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, Software Visualization, chapter 2, pp. 29--34. MIT Press, 1998.

27. Myers, B. A. 1986. Visual programming, programming by example, and program visualization: a taxonomy. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Boston, Massachusetts, United States, April 13 - 17, 1986). M. Mantei and P. Orbeton, Eds. CHI '86. ACM Press, New York, NY, 59-66.

28. Brad A. Myers. Taxonomies of Visual Programming and Program Visualization, Journal of Visual Languages and Computing. vol. 1, no. 1. March 1990, pp. 97-123.

29. Brooks Frederick, The Mythical Man-Month: essays on Software Engineering, published by Addison Wesley Longman Inc, September 2004(24th edition), ISBN 0201835959.

30. 17 USC § 117.

31. IEEE Std 610.12 – 1990.

32. IEC1131-1, 1992; IEC 902, 1987.

33. Musa, J., Iannino, A., and Okumoto, K. (1987). Software Reliability, Measurement, Prediction, Application. McGraw-Hill Book Co., New York.

34. Smith, D.J., and Wood, K.B.: `Engineering Quality Software: a review of current practices, standards and guidelines including new methods and development tools', 2nd edition (Elsevier Applied Science, 1989.

35. Turing, A.M. (1950). Computing machinery and intelligence. Mind, 59, 433-460.

36. Tukey J.W., "The Teaching of Concrete Mathematics," American Mathematical Monthly, 65, No. 1. (January 1958), p. 2.

37. The American Heritage® Dictionary of the English Language, Third Edition Copyright © 1992, 1996, by Houghton Mifflin Company.

38. Software Engineering: A practitioner’s Approach by, published by McGraw-Hill Professional Mar 1, 2004 ISBN: 007301933X.

39. UK MoD, Def Stan 00-55,1997.

40. ESA, ECSS-P-001A, 1997.

41. Introduction to Algorithms (Second Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, published by MIT Press and McGraw-Hill.

42. Introduction to computing and algorithms by R.L. Shackelford , published by Addison Wesley Longman Inc, 1998 , ISBN 0201314517.

43. Price B., Baecker R., and Small I., An Introduction to Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds), Software Visualization, chapter 1, pp. 3--27. MIT Press, 1998.

44. ANSI/IEEE Std 1084-1 986 IEEE Standard Glossary of Mathematics of Computing Terminology.

45. Compact Oxford Dictionary, publicly available online at http://www.askoxford.com/concise_oed/algorithm?view=uk September 2006.

46. Oxford English Dictionary Online (subscription required).

47. The American Heritage® Dictionary of the English Language: Fourth Edition. 2000.

48. Merriam-Webster Online Dictionary, 2006.

49. MCCORMICK, B. H., DEFANTI, T. A., AND BROWN, M. D. Visualization in scientific computing---A synopsis. Computer Graphics & Applications 7, 7 (1987), 61—70.

50. Price B., Baeker R.M., Small I., A Principled Taxonomy of Software Visualization. Journal of Visual Languages and Computing, Vol.4 No.3, pp.211-266, September 1993.

51. Roman, G. and Cox, K. C. 1992. Program visualization: the art of mapping programs to pictures. In Proceedings of the 14th international Conference on Software Engineering (Melbourne, Australia, May 11 - 15, 1992). ICSE '92. ACM Press, New York, NY, 412-420.

52. Diehl, S. 2005. Software visualization. In Proceedings of the 27th international Conference on Software Engineering (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM Press, New York, NY, 718-719.

53. Oudshoorn M. Widjaja H.W., Ellershaw S., Aspects and Taxonomy of Program Visualisation. In P. Eades and K. Zhang (editors), Software Visualisation, Chapter 1. World Scientific Press, Singapore, 1996.

54. Ellershaw, S. & Oudshoorn, M. (1994), Program visualization - the state of the art, Technical Report TR 94-19, Department of Computer Science, University of Adelaide.

55. Baecker R.M., Enhancing Program Readability and Comprehensibility with Tools for Program Visualization, in Proceedings of the 10th International Conference on Software Engineering, pp. 356-366, April 1988.

56. Petre M., Blackwell A.F., and Green T.R.G., Cognitive Questions in Software Visualisation, In J. Stasko, J. Domingue, B. Price, and M. Brown (Eds.), Software Visualization: Programming as a Multi-Media Experience, MIT Press, January 1998.

57. Brown M.H., Perspectives on algorithm animation, Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 33-38, May 15-19, 1988, Washington, D.C., United States.

58. Kerren A. and Stasko J.T., Algorithm Animation - Chapter Introduction. In S. Diehl (ed): Software Visualization, volume 2269 of LNCS pages 1-15. Springer, 2001.

59. Burnett M., Visual Programming. Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), John Wiley & Sons Inc., New York, (1999).

60. Golin E. J. & Reiss S. P., The Specification of Visual Language Syntax, IEEE Workshop on Visual Languages, (1990), pp.105-110.

61. McIntyre D. W. and Glinert E. P., Visual Tools for Generating Iconic Programming Environments, (1992).

62. Stanford's Encyclopedia for Philosophy http://plato.stanford.edu/entries/aristotle-metaphysics/#Cat September 2006.

63. Wikipedia, On line Encyclopedia, http://en.wikipedia.org/wiki/Taxonomy September 2006.

64. ANSI/IEEE Std 1002-1987.

65. What is?com. IT on-line encyclopedia, http://whatis.techtarget.com/definition/0,,sid9_gci331416,00.htm September 2006.

66. Dictionary of Computing, Oxford:Oxford University Press,1983.

67. Price B., Baeker R.M., Small I., A taxonomy of software visualization. In Proceedings of the 25th Hawaii International Conference on System Sciences, volume II, pages 597--606, Kauai, HI, January 1992.

68. Eisenstadt, M.; Domingue, J.; Rajan, T.; Motta, E., "Visual knowledge engineering," Software Engineering, IEEE Transactions on, vol.16, no.10pp.1164-1177, October 1990.

69. N. C. Shu, Visual programming, Van Nostrand Reinhold Co., New York, NY, 1988.

70. Stasko, J. T. & Patterson, C. (1992). Understanding and Characterizing Software Visualization Systems. In Proceedings of IEEE 1992 Workshop on Visual Languages, (pp. 3-10) New York, 15-18 September 1992 : IEEE Computer Society Press.

71. Baecker R.M., 1986, An Application Overview of Program Visualization. Computer Graphics: SIGGRAPH '86, 20 (4):325, July 1986.

72. Tilley, S. and Huang, S. 2002. Documenting software systems with views III: towards a task-oriented classification of program visualization techniques. In Proceedings of the 20th Annual international Conference on Computer Documentation (Toronto, Ontario, Canada, October 20 - 23, 2002). SIGDOC '02. ACM Press, New York, NY, 226-233.

73. Maletic, J. I., Marcus, A., and Collard, M. L. 2002. A Task Oriented View of Software Visualization. In Proceedings of the 1st international Workshop on Visualizing Software for Understanding and Analysis (June 26 - 26, 2002). VISSOFT. IEEE Computer Society, Washington, DC, 32.

74. Card, S. K., Mackinlay, J., and Shneiderman, B., Readings in Information Visualization Using Vision to Think, San Francisco, CA, Morgan Kaufmann, 1999.

75. Baecker R.M., Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, Software Visualization: Programming as a Multimedia Experience, chapter 24, pp. 369–381. MIT Press, Cambridge, MA, 1998.

76. Brown, M. H. & Sedgewick, R. (1984a). “Progress Report: Brown University Instructional Computing Laboratory.” ACM SIGCSE Bulletin, 16(1): 91-101.

77. Brown M.H., Sedgewick R., Interesting events. In Stasko et al. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), Software Visualization: Programming as a Multimedia Experience. MIT Press, 1998, chapter 12, pp. 155—171.

78. Brown, M. H. (1988a). Algorithm Animation. New York: MIT Press.

79. Brown, M. H. (1988b). “Exploring Algorithms Using Balsa II.” IEEE Computer, 21(5): 14-36.

80. Brown, M. H. (1998). A taxonomy of algorithm animation displays. In J. T. Stasko et al. (Eds.), Software visualization (pp. 35-12). Cambridge, MA: MIT Press.

81. Stasko, J.T. (1990). "Tango: A Framework and System for Algorithm Animation" IEEE Computer 23, September 1990, pp. 27-39.

82. Stasko, J.T., The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces, Journal of Visual Languages and Computing, Volume 1, Number 3, September 1990, pp. 213-236.

83. Stasko, J.T., (1992). "Animating algorithms with XTANGO" SIGACT News 23, Spring 1992, pp. 67-71.

84. Georgia’s Institute of Technology site http://www-static.cc.gatech.edu/gvu/softviz/algoanim/bpack.gif September 2006.

85. Georgia’s Institute of technology site, http://www-static.cc.gatech.edu/gvu/softviz/algoanim/xtango.html September 2006.

86. Stasko,J.T and Wehrli J.F., "Three-dimensional computation visualization," Tech. Rep. GIT-GVU-92-20, Georgia Institute of Technology, 1992.

87. Stasko, J.T., Kraemer E. “A Methodology for Building Application-Specific Visualizations of Parallel Programs”, Journal of Parallel and Distributed Computing, Vol 18, No 2, June 1993, pp. 258-264.

88. Stasko, J.T., -Notes from CS7450 Information Visualization Course at Georgia Tech College of Computing (Spring 2005) http://www-static.cc.gatech.edu/classes/AY2005/cs7450_spring/Talks/19-softvis.pdf, September 2006.

89. Stasko, J.T., "Using student-built animations as learning aids," in Proceedings of the ACM Technical Symposium on Computer Science Education. New York: ACM Press, 1997, pp. 25-29.

90. Georgia's Institute of Technology site http://www-static.cc.gatech.edu/gvu/softviz/ September 2006.

91. Georgia's Institute of Technology site, http://www-static.cc.gatech.edu/gvu/softviz/parviz/polkaanims.html September 2006.

92. Brown, M. H. (1991). "Zeus: A System for Algorithm Animation and Multi-View Editing. "Proceedings of IEEE Workshop on Visual Languages, New York: IEEE Computer Society Press: 4-9.

93. Brown, M. H. (1992). Zeus: A System for Algorithm Animation and Multi-view Editing (Research Report No. 75). DEC Systems Research Center, Palo Alto, CA.

94. Brown, M. H and J. Hershberger. Color and sound in algorithm animation. Computer, Volume 25, Issue 12, Dec. 1992 pp. 52 – 63.

95. Brown, M. H. and Najork, M. A. 1993. Algorithm animation using 3D interactive graphics. In *Proceedings of the 6th Annual ACM Symposium on User interface Software and Technology* (Atlanta, Georgia, United States). UIST '93. ACM Press, New York, NY, 93-100).

96. Eick, S. G., Steffen, J. L., Sumner, E. E. SeeSoft --A Tool for Visualizing Line Oriented Software Statistics. IEEE Trans. on Software Engineering, 18(11),1992, 957 – 968.

97. BAKER, M.J., EICK, S.G. 1995. Space-filling software visualization. Journal of Visual Languages and Computing 6, 2, 119—133.

98. Johnson B., Shneiderman B., Tree-maps: A spacelling approach to the visualization of hierarchical information structures. In IEEE Visualization '91 Conference Proceedings, pages 284{291, San Diego, California, October 1991.

99. Ball, T. A. and Eick, S. G. (1996). Software visualization in the large. *IEEE Computer*, 29(4):33-43.

100. Ball T, Eick S, Mockus A., Web-based Analysis of large Scale Software Systems, http://www.mockus.us/papers/websoft/index.html September 2006.

101. Voinea, L., Telea, A., and van Wijk, J. J. 2005. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (St. Louis, Missouri, May 14 - 15, 2005). SoftVis '05. ACM Press, New York, NY, 47-56.

102. Baecker, R.M. and Marcus, A., Printing and Publishing C Programs. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), Software Visualization: Programming as a Multimedia Experience. MIT Press, 1998, pp. 45-61.

103. Storey, M.A.D. Muller, H., Manipulating and documenting software structures using shrimp views, International Conference in Software Maintenance, IEEE Computer Society Press, 1995, pp. 275—285.

104. Storey, M.A.D, Wong, K., Fracchia, F., Muller ,H., On integrating visualization techniques for effective software exploration. In Proceedings of the IEEE Symposium on Information Visualization, IEEE Visualization, 1997.

105. Wu, J. and Storey, M. D. 2000. A multi-perspective software visualization environment. In Proceedings of the 2000 Conference of the Centre For Advanced Studies on Collaborative Research (Mississauga, Ontario, Canada, November 13 - 16, 2000). S. A. MacKay and J. H. Johnson, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 15.

106. Storey, M.A., Best, C., Michaud, J., Rayside, D., Litoiu, M.,Musen, M. SHriMP Views: an Interactive Environment for Information Visualization and Navigation. Proc. CHI ‘02, ACM Press, NY, 520 – 521.

107. Lintern, R., Michaud, J., Storey, M., and Wu, X. 2003. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In Proceedings of the 2003 ACM Symposium on Software Visualization (San Diego, California, June 11 - 13, 2003). SoftVis '03. ACM Press, New York, NY, 47-ff.

108. Lethbridge, T. and Singer, J. 1997. Understanding Software Maintenance Tools: Some Empirical Research. In Proceedings of the IEEE Workshop on Empirical Studies of Software Maintenance (WESS97), Bari, Italy, 157-162.

109. Brown M.H., Hershberger J., Program Auralization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, Software Visualization, chapter 10, pages 137-142. MIT Press, 1998.

110. Kobsa A., Wahlster,W., editors. Computational Linguistics, volume 14(3). MIT Press for the Association of Computational Linguistics, September 1988. Special Issue on User Modeling.

111. User Modeling Inc. web page http://www.um.org/, September 2006.

112. Kobsa A., A taxonomy of beliefs and goals for user models in dialog systems. In Alfred Kobsa and Wolfgang Wahlster, editors, User Models in Dialog Systems, pages 52--73. Springer-Verlag, Berlin, 1989.

113. MIT Media Lab- Affective Computoing group http://affect.media.mit.edu/areas.php?id=understanding , September 2006.

114. Wenger, E., Artificial Intelligence and Tutoring Systems. 1987, Los Altos, California: Morgan Kaufmann Publishers.

115. Motta,E., Eisenstadt, M., Pitman, K., West,M., “Support for knowledge acquisition in the knowledge engineers’ assistant (KEATS),” Expert Syst., vol. 5, pp. 6-28, 1988.

116. Storey, M. D., Fracchia, F. D., and Mueller, H. A. 1997. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. In *Proceedings of the 5th international Workshop on Program Comprehension (WPC '97)* (May 28 - 30, 1997). WPC. IEEE Computer Society, Washington, DC, 17.

117. Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. 1997. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre For Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada, November 10 - 13, 1997). J. H. Johnson, Ed. IBM Centre for Advanced Studies Conference. IBM Press, 21.

118. IEEE's Software Engineering Body of Knowledge @ www.swebok.org.

119. Bergeron, D., 1993. Visualization Reference Models (Panel Position Statement), Proceedings of IEEE Visualization, G.M. Nielson and D. Bergeron (Eds), IEEE Computer Society Press.

120. Wehrend, S. and Lewis, C. 1990. A problem-oriented classification of visualization techniques. In *Proceedings of the 1st Conference on Visualization '90* (San Francisco, California, October 23 - 26, 1990). A. Kaufman, Ed. IEEE Visualization. IEEE Computer Society Press, Los Alamitos, CA, 139-143.

121. Stephen M. Casner, A Task-Analytic Approach to the Automated Design of Graphic Presentations, ACM Trans Graphics, Vol. 10, No. 2, April 1991, pp. 111-151.

122. Shneiderman, B., "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations", in Proceedings of IEEE Visual Languages, 1996, pp. 336-343.

123. Wiss, U., Carr, D., and Jonsson, H., "Evaluating Three- Dimensional Information Visualization Designs A Case Study of Three Designs", in Proceedings of International Conference on Information Visualisation, London, England, July 29-31 1998.

124. Zhou, M.X., Feiner, S.K., Visual task characterization for automated visual discourse synthesis, Proceedings of the ACM CHI '98 Conference on Human Factors in Computing Systems, 1998, pp. 392 -- 399, Los Angeles, California, USA.

125. Norman, D. A., The Psychology of Everyday Things. Basic Books, New York, 1988.

126. Zhou, M. X. and Feiner, S. K. 1997. Top-down hierarchical planning of coherent visual discourse. In Proceedings of the 2nd international Conference on intelligent User interfaces (Orlando, Florida, United States, January 06 - 09, 1997). J. Moore, E. Edmonds, and A. Puerta, Eds. IUI '97. ACM Press, New York, NY, 129-136.

127. MacKinlay, J. D., "Automating the design of graphical presentation of relational information", ACM Transaction on Graphics, vol. 5, no. 2, April 1986, pp. 110-141.

128. Russell, D. M., Stefik, M. J., Pirolli, P., and Card, S. K. 1993. The cost structure of sensemaking. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Amsterdam, The Netherlands, April 24 - 29, 1993). CHI '93. ACM Press, New York, NY, 269-276.

129. Lohse, G. L., Biolsi, K., Walker, N., and Rueter, H. H. 1994. A classification of visual representations. Commun. ACM 37, 12 (December 1994), 36-49.

130. Roth, S.F. and Mattis, J. (1990) Data characterization for intelligent graphics presentation. In Proceedings of CHI90 (Seattle, April 1-5, 1990) New York: ACM, 193-200.

131. Koschke, R., Software Visualization for Reverse Engineering In S. Diehl (ed): Software Visualization, volume 2269 of LNCS pp. 138-150. Springer, 2001.

132. Duke, D.J.; Brodlie, K.W.; Duce, D.A.; Herman, I., "Do you see what I mean? [Data visualization]," Computer Graphics and Applications, IEEE , vol.25, no. 3, pp. 6- 9, May-June 2005.

133. Report of the Visualization Ontology Workshop, http://www.nesc.ac.uk/talks/393/vis_ontology_report.pdf, September 2006.

134. Haber R.B., McNabb, D.A., "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems," Visualization in Scientific Computing, G.M. Nielson, B. Schriver, and L.J. Rosenblum, eds., IEEE CS Press, 1990.

135. Robertson P., De Ferrari, L., Systematic Approaches to Visualization: Is a Reference Model Needed? in Scientific Visualization, 1994, Advances and Challenges, Ed: L. Rosenblum, R.A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, D. Thalmann , Academic Press.

136. Brodlie,K., Duce,D., Gallop,J., Sagar,M., Walton,J., Wood,J., Visualization in Grid Computing Environments. Proceedings of IEEE Visualization 2004, edited by Holly Rushmeier, Greg Turk and Jarke J. van Wijk, pp. 155-162. ISBN:0-7803-8788-0.

137. Nielson, G.M., Visualization in Scientific and Engineering Computation. IEEE Computer, 24(9), pp. 58-66, September 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Thomas Otani
   Naval Postgraduate School
   Monterey, California

4. Dr. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

5. Dimitrios Spyrou
   Naval Postgraduate School
   Monterey, California